Notes du cours de Calculabilité et Logique première partie 2006 - 2009

H. Comon-Lundh (06-07)
P. Schnoebelen (07-08)
S. Haddad (08-09)
P.-A. Reynier (06-07)
F.-R. Sinot (07-08)
C. Sirangelo (08-09)

Table des matières

Chapitre 1

Introduction

On considère un problème $P(d)$ qui dépend d'une donnée d (par exemple : est-ce que le nombre d est premier ? Est-ce que le graphe d est planaire ? Est-ce que le programme d finit toujours par s'arrêter, ceci quel que soit le nombre n qu'on lui soumet en input ? etc.).

D'un point de vue informatique, résoudre $P(d)$ suppose d'une part un modèle de calcul M et d'autre part un programme dans ce modèle, dont l'input est d et qui s'arrête en répondant à la question. Le problème peut être résolu (on dit qu'il est *décidable*) si l'on peut trouver un modèle de calcul et un programme qui répond à la question (pour toute donnée d). La *thèse de Church* stipule que cette question de décidabilité est indépendante du modèle de calcul et, jusqu'à récemment, aucun modèle de calcul n'est venu invalider cette thèse. Durant le cours nous rencontrerons plusieurs modèles de calcul et montreront qu'ils sont équivalents (simulables les uns par les autres en un sens qui sera précisé). Ainsi, la notion de décidabilité sera définie pour un modèle de calcul particulier (les machines de Turing) mais elle a une portée universelle.

Une autre question fondamentale est la mesure de la difficulté d'un problème P, lorsqu'il est décidable. La complexité de la solution se mesure en termes de ressources utilisées par un programme résolvant P : nombre d'étapes de calcul (complexité en temps), espace de stockage nécessaire (complexité en espace), taille du programme, ...

De ce point de vue, les modèles de calcul ne sont pas équivalents. On aimerait définir la complexité intrinsèque d'un problème (en temps par exemple) dans un modèle M, comme le minimum des temps d'exécution pour tous les programmes résolvant M. Mais le temps de calcul d'un programme qui résout P dépend de la valeur de la donnée d. Il faut donc comparer des fonctions, mais l'ordre canonique défini par « $f \geq g \overset{\text{def}}{\Leftrightarrow} \forall d. f(d) \geq g(d)$ » n'est ni total, ni bien fondé.

Chapitre 2

Machines de Turing et récursivité

Notations
Un *mot* sur l'alphabet Σ est une suite (éventuellement vide) d'éléments de Σ. Sauf précision contraire, on ne considère que des mots finis.

Si $A, B \subseteq \Sigma^*$ sont des ensembles de mots sur l'alphabet Σ,
- $A + B$ est l'ensemble $A \cup B$;
- a (où $a \in \Sigma$) est l'ensemble réduit au mot a ;
- ϵ est le mot vide ;
- $A \cdot B$ est l'ensemble $\{w_1 w_2 \mid w_1 \in A, w_2 \in B\}$;
- A^* est l'ensemble des mots w tels qu'il existe un entier $k \in \mathbb{N}$ et k mots $w_1, \ldots, w_k \in A$ tels que $w = w_1 \cdots w_k$ ($k = 0$ est possible).

2.1 Machines de Turing

Il existe de très nombreux modèles de calcul, représentant plus ou moins bien des architectures matérielles, des circuits ou des langages. Les *machines de Turing* constituent néanmoins le modèle de référence incontournable, pour des raisons historiques. Il s'agit d'un modèle de très bas niveau dans lequel on ne dispose que d'une structure de données (les mots) et, d'une seule instruction (GOTO) en dehors des lectures et écritures.

Il existe plusieurs variantes des machines de Turing. Elles sont toutes équivalentes (en accord avec la thèse de Church) du point de vue de l'expressivité. Mais elles ne sont pas équivalentes du point de vue de la complexité. La définition 2.1.1 qui suit sera modifiée plus tard quand on définira les classes de complexité.

Définition 2.1.1 *Une* machine de Turing *(MT) est un tuple* $(Q, q_0, \Sigma, \delta, \{B, \$\}, q_B)$ *où*
- Q *est un ensemble fini d'*états*,*
- $q_0 \in Q$ *est l'*état initial*,*
- Σ *est un* alphabet *fini,*
- $B, \$ \in \Sigma$ *sont deux symboles spéciaux distincts (B = « blanc », $\$$ = « marqueur de début »),*
- Q_B *est un sous-ensemble d'états qui autorise l'écriture de blancs,*
- $\delta : Q \times \Sigma \mapsto \big(Q \uplus \{\textbf{\textit{accept}}, \textbf{\textit{reject}}\}\big) \times \Sigma \times \{\leftarrow, \rightarrow, \downarrow\}$ *est une* fonction de transition *vérifiant les contraintes suivantes :*
  1. *Pour tout $q \in Q$, il existe $q' \in Q$ tel que $\delta(q, \$) = (q', \$, \rightarrow)$*
  2. *Pour tout $q, q' \in Q$, $x \in \Sigma$, $d \in \{\leftarrow, \rightarrow, \downarrow\}$ $\delta(q, x) = (q', \$, d) \Rightarrow x = \$ \wedge d = \rightarrow$*
  3. *Pour tout $x \neq \$$, il existe q' tel que $\delta(q_B, x) = (q', B, \leftarrow)$*
  4. *Pour tout $q, q' \in Q$, $x \in \Sigma$, $d \in \{\leftarrow, \rightarrow, \downarrow\}$ $\delta(q, x) = (q', B, d) \Rightarrow q' \in Q_B \wedge d = \leftarrow$*
  5. *Pour tout $q \in Q, q' \in Q_B$, $x, y \in \Sigma$, $d \in \{\leftarrow, \rightarrow, \downarrow\}$*
     $\delta(q, x) = (q', y, d) \Rightarrow x = y = B \wedge d = \leftarrow$

6. *Pour tout* $q, q' \in Q,\ x \in \Sigma\ \delta(q, B) = (q', x, \rightarrow) \Rightarrow x \neq B$

Quelques remarques et notations en vrac :

1. On notera aussi **accept** (resp. **reject**) q_Y (resp. q_N). On notera aussi q_0, q_I.
2. δ est une *fonction* et pas forcément une application. Son domaine de définition peut n'être qu'un sous-ensemble strict de $Q \times \Sigma$. Dans la suite, on fera comme si δ était une application en complétant la définition de δ par $\delta(q, a) \stackrel{\text{def}}{=} (\textbf{reject}, a, \downarrow)$ sur les couples (q, a) où elle est indéfinie.
3. Les restrictions sur δ impliquent qu'on ne se déplace jamais à gauche du marqueur de début et qu'on ne peut pas l'effacer. Elles impliquent aussi que le contenu de la bande a toujours la forme d'un \$, suivi de lettres de $\Sigma \setminus \{B, \$\}$ et terminé par des blancs.
4. Cette définition correspond aux machines *à un seul ruban* appelé encore *bande* : les machines à plusieurs rubans seront abordées ci-dessous.
5. Il n'y a pas d'état final dans cette définition. On pourrait aussi considérer **accept** comme unique état final ; il n'y a pas de transition depuis un état final.
6. Cette définition autorise de ne pas faire de mouvement, c.-à-d. qu'elle autorise $\delta(q, a) = (\ldots, \ldots, \downarrow)$, ce qui n'est pas toujours permis par d'autres auteurs.
7. On trouve parfois des définitions où figurent deux alphabets : un alphabet « de travail » et un alphabet « d'entrée ». Pour les questions que nous abordons ici, cette distinction n'est pas pertinente.
8. Nos machines sont *déterministes* : δ est une fonction et non une relation. Pour l'heure, cela nous suffit. Les machines non-déterministes seront introduites quand cela sera nécessaire.

La définition suivante donne la sémantique d'un calcul de la machine de Turing où la bande est vue comme deux mots finis dont la concaténation est le contenu significatif de la bande complété éventuellement par un ou plusieurs blancs. Le second mot est non vide et la tête de lecture pointe sur sa première lettre. La configuration est alors un triplet composé des deux mots et d'un état.

Définition 2.1.2 *1. Une* configuration *de la machine* $M = (Q, q_0, \Sigma, \delta, \{B, \$\})$ *est un triplet* $\gamma = (w, q, w')$ *où* $w, w' \in \Sigma^*, q \in Q \cup \{\textbf{\textit{accept}}, \textbf{\textit{reject}}\}$ *et* $w' \neq \epsilon$.

2. Étant donné $w_0 \in \Sigma^*$, *la* configuration initiale sur l'entrée *(ou la* donnée*)* w_0 *est* $(\epsilon, q_0, \$w_0)$.

3. Les configuration finales *sont celles de la forme* (w, q, w') *telles que* $q \in \{\textbf{\textit{accept}}, \textbf{\textit{reject}}\}$.

4. M peut faire un mouvement *de* (w, q, aw') *vers* (w_1, q_1, w_1'), *ce que l'on note* $(w, q, aw') \vdash (w_1, q_1, w_1')$ *ssi on est dans l'un des cas suivants :*
- $w_1 = wb, w_1' = w'B$ *si* $\delta(q, a) = (q_1, b, \rightarrow)$;
- $w_1 = w, w_1' = bw'$ *si* $\delta(q, a) = (q_1, b, \downarrow)$;
- $w = w_1 c$, $w_1' = cbw'$ *si* $\delta(q, a) = (q_1, b, \leftarrow)$.

Les mouvements gauche et droit sont représentés dans la figure 2.1.

Définition 2.1.3 *Un* calcul *d'une machine M sur un mot w est une suite de configurations* $\gamma_0, \gamma_i, \ldots, \gamma_n, \ldots$ *telle que* $\gamma_0 = (\epsilon, q_0, \$w)$ *et* $\forall i > 0.\ \gamma_{i-1} \vdash \gamma_i$. *De plus, cette suite ne peut être finie que si elle s'achève sur une configuration finale.*

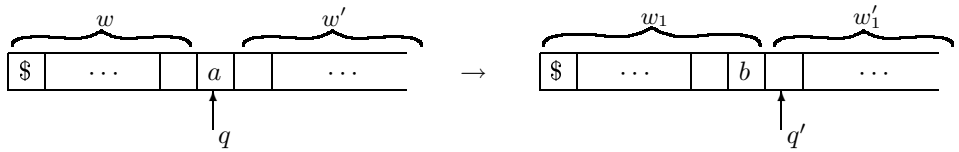
Exemple 2.1.1 *Soit M la machine dont la fonction de transition est donnée par la table :*

	\$	0	1	B
q_0	$q_0, \$, \rightarrow$	$q_0, 0, \rightarrow$	$q_1, 0, \rightarrow$	$\textbf{\textit{accept}}, 0, \downarrow$
q_1		$q_0, 1, \rightarrow$	$q_1, 1, \rightarrow$	$\textbf{\textit{accept}}, 1, \downarrow$

On déroulera un calcul de la machine sur le mot d'entrée 0110.

Définition 2.1.4 *Une* machine de Turing à k rubans *est un tuple* $(Q, q_0, \Sigma, \delta, \{B, \$\})$ *où*

Mouvement droit



Mouvement gauche

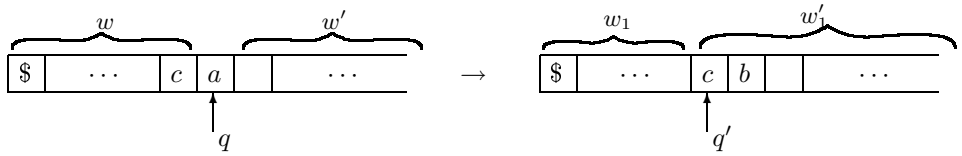


FIG. 2.1 – Configurations et mouvements d'une machine de Turing

- *Q est un ensemble fini d'états*
- *$q_0 \in Q$ est l'état initial*
- *Σ est un ensemble fini de symboles*
- *$B, \$ \in \Sigma$*
- *Pour tout $i \leq k$, $Q_{B,i}$ est un sous-ensemble d'états qui permet l'écriture de B sur le ruban i*
- *δ est une application de $Q \times \Sigma^k$ dans $(Q \cup \{\textbf{\textit{accept}}, \textbf{\textit{reject}}\}) \times (\Sigma \times \{\leftarrow, \downarrow, \rightarrow\})^k$ vérifiant des contraintes similaires à celles d'une machine à un ruban.*

La définition de configuration s'étend à k rubans : il suffit de considérer cette fois les k contenus des rubans et les k positions des têtes de lecture. La configuration initiale ne contient de symboles non blanc ou \$ que sur le premier ruban. La définition de mots acceptés, rejetés, etc., s'étend. Pour le calcul des fonctions, on distingue un *ruban d'entrée* contenant la donnée et un *ruban de sortie* contenant le résultat.

Exemple 2.1.2 *Nous donnons, sous la forme d'une table, la fonction de transition d'une machine à deux rubans qui additionne deux nombres écrits en binaire de droite à gauche.*

	$\$,\$$	$0,\$$	$1,\$$	$B,\$$	$0,B$	$1,B$	B,B	$0,0$	$0,1$	$1,0$	$1,1$	$B,0$	$B,1$
q_0	q_0 $\$,\rightarrow$ $\$,\downarrow$	q_0 $0,\rightarrow$ $\$,\downarrow$	q_0 $1,\rightarrow$ $\$,\downarrow$	q_1 $B,\rightarrow$ $\$,\rightarrow$	q_0 $B,\rightarrow$ $0,\rightarrow$	q_0 $B,\rightarrow$ $1,\rightarrow$	q_1 $B,\leftarrow$ $B,\leftarrow$						
q_1	q_2 $\$,\rightarrow$ $\$,\rightarrow$	q_1 $\$,\leftarrow$ $\$,\downarrow$	q_1 $\$,\leftarrow$ $\$,\downarrow$					q_1 $0,\leftarrow$ $0,\leftarrow$	q_1 $0,\leftarrow$ $1,\leftarrow$	q_1 $1,\leftarrow$ $0,\leftarrow$	q_1 $1,\leftarrow$ $1,\leftarrow$	q_1 $B,\leftarrow$ $0,\leftarrow$	q_1 $B,\leftarrow$ $1,\leftarrow$
q_2					q_2 $0,\rightarrow$ $B,\downarrow$	q_2 $1,\rightarrow$ $B,\downarrow$		q_2 $0,\rightarrow$ $0,\rightarrow$	q_2 $0,\rightarrow$ $1,\rightarrow$	q_2 $1,\rightarrow$ $0,\rightarrow$	q_3 $0,\rightarrow$ $0,\rightarrow$	q_2 $0,\rightarrow$ $B,\rightarrow$	q_2 $1,\rightarrow$ $B,\rightarrow$
q_3					q_2 $1,\rightarrow$ $B,\downarrow$	q_3 $0,\rightarrow$ $B,\downarrow$		q_2 $1,\rightarrow$ $0,\rightarrow$	q_3 $0,\rightarrow$ $1,\rightarrow$	q_3 $0,\rightarrow$ $0,\rightarrow$	q_3 $1,\rightarrow$ $0,\rightarrow$	q_2 $1,\rightarrow$ $B,\rightarrow$	q_3 $0,\rightarrow$ $B,\rightarrow$

Nous anticipons les notions de complexité pour indiquer que la prise en compte de la complexité spatiale nécessite de distinguer les entrées et les sorties des données de travail.

Définition 2.1.5 *Une machine de Turing I/O est une machine à $k+2$ rubans dont le premier ruban est appellé le ruban d'entrée et le dernier ruban est appelé le ruban de sortie qui vérifie :*
- *on n'écrit jamais*[1] *sur le ruban de lecture ;*
- *les transitions ne dépendent jamais du contenu du ruban d'écriture.*

[1] Il s'agit d'un abus de langage puisque une MT écrit toujours un symbole sur le ruban, lors de chaque transition. Simplement on veut dire qu'elle écrit le symbole qu'elle vient de lire et que le contenu du ruban n'est jamais modifié.

De cette façon, on peut avoir des machines de Turing travaillant avec une taille d'espace, fonction logarithmique de la taille de l'entrée. Ceci sera formalisé bientôt.

2.2 Programmer avec des machines de Turing ($LWHILE$)

Les machines de Turing constituent un modèle de bas niveau. Cependant même avec un modèle de si bas niveau, il est possible de fabriquer rapidement un langage de programmation pour les entiers.

Tout d'abord, chaque ruban correspondra à une variable entière codée en binaire dont le bit de poids le plus fort est situé à droite du \$. Donc un mot infini sur le ruban représentant un entier s'écrit $\$(0+1(0+1)^*)B^\infty$

Afin d'obtenir un langage de programmation, il nous faut implémenter les instructions élémentaires (copie, incrémentation, décrémentation) et les structures de contrôle : if, $while$ et la concaténation. Nous appelons ce langage $LWHILE$.

Pour les structures de contrôle, on suppose que l'on a déjà traduit les blocs de la structure par des machines de Turing toutes travaillant sur les mêmes bandes. Chaque machine de Turing lorsqu'elle se termine renvoie vrai si elle se termine dans l'état q_Y et faux dans l'état q_N.

Nous nous bornerons à développer deux cas :

- l'instruction élémentaire $x := x-1$ qui si $x = 0$ renvoie vrai et sinon décrémente x et renvoie faux.
- la boucle $while\ MT_1\ do\ MT_2$ qui exécute la machine MT_2 tant que la machine MT_1 renvoie vrai.

La machine de Turing correspondant à $x := x - 1$ commence par tester si $x = 0$:

$$\delta(q_I, 0) = (q_Y, 0, \downarrow),\ \delta(q_I, 1) = (q_s, 1, \rightarrow)$$

Dans le cas contraire, elle va au bout de l'entier :

$$\forall x \in \{0,1\}\ \delta(q_s, x) = (q_s, x, \rightarrow), \delta(q_s, B) = (q_d, B, \leftarrow)$$

Puis elle décrémente avec retenue :

$$\delta(q_d, 1) = (q_e, 0, \leftarrow), \delta(q_d, 0) = (q_d, 1, \leftarrow)$$

Elle s'arrête à gauche ... :

$$\forall x \in \{0,1\}\ \delta(q_e, x) = (q_e, x, \leftarrow), \delta(q_d, \$) = (q_t, \$, \rightarrow)$$

Et repart à droite pour savoir s'il faut enlever le 0 de tête :

$$\forall x \in \{0,1\}\ \delta(q_t, 1) = (q_N, 1, \downarrow), \delta(q_t, 0) = (q'_s, 0, \rightarrow)$$

Dans le cas d'une normalisation (nombre $01111\ldots1$), elle va au bout de l'entier :

$$\forall x \in \{0,1\}\ \delta(q'_s, x) = (q'_s, x, \rightarrow), \delta(q'_s, B) = (q'_d, B, \leftarrow)$$

Puis recopie les 1 sur sa droite :

$$\delta(q'_d, 1) = (q_1, B, \leftarrow)$$
$$\delta(q_1, 0) = (q_1, 1, \leftarrow), \delta(q_1, 1) = (q_1, 1, \leftarrow), \delta(q_1, \$) = (q_N, \$, \leftarrow)$$

La machine de Turing implémentant le $while$ fonctionne ainsi.

- Elle inclut des copies des machines MT_1 et MT_2 de telle façon que les ensembles d'états soient disjoints.
- Son état initial est l'état initial de MT_1.
- Les transitions vers q_Y de MT_1 sont redirigées vers l'état initial de MT_2.
- Les transitions vers q_Y, q_N de MT_2 sont redirigées vers l'état initial de MT_1.

Ainsi il est maintenant immédiat d'effectuer une addition, une mutiplication, etc, uniquement à partir de décrémentations et d'incrémentations. Ce court développement est une illustration de la thèse de Church.

2.3 Fonctions calculables, langages décidables, complexité de calcul

Si la machine M s'arrête sur la donnée x, soit (w_1, q, w_2) la configuration finale ($q \in \{\textbf{accept}, \textbf{reject}\}$). On note $M(x)$ le mot obtenu en retirant à w_1w_2 le \$ de tête et les blancs de fin de mot. Si M ne

s'arrête pas sur la donnée x, par convention, $M(x) = \perp$.

Définition 2.3.1 *Si f est une application de $D \subseteq (\Sigma \setminus \{B, \$\})^*$ dans Σ^*, M* calcule *f si, pour tout $w \in D$, $M(w) = f(w)$. Si une telle machine de Turing existe, f est dite* calculable.

Observons qu'une telle machine de Turing s'arrête toujours (au moins pour les entrées licites).

La notion de fonction calculable s'étend aux entiers. Les entiers sont codés en base 2 par des mots de $0 + 1(0 + 1)^*$; on note $\overline{n}$ le codage de $n \in \mathbb{N}$. Une fonction f de $\mathbb{N}$ dans $\mathbb{N}$ est *calculable* si la fonction qui, pour tout n dans le domaine de définition de f, associe à $\overline{n}$ le mot $\overline{f(n)}$ est calculable.

Pour les fonctions à deux arguments (de $\Sigma^* \times \Sigma^*$ dans Σ^*), on se ramène aux fonctions de $(\Sigma \uplus \{\#\})^*$ dans $(\Sigma \uplus \{\#\})^*$, définies seulement pour les mots $w\#w'$.

Proposition 2.3.1 *Quand on représente les entiers naturels par des mots du langage $0+1(0+1)^*$, les fonctions arithmétiques suivantes sont calculables :*

- $(n, m) \mapsto n + m$;
- $(n, m) \mapsto n \times m$;
- $(n, m) \mapsto n^m$ *(fonction non définie pour $n = m = 0$).*

Preuve
La preuve est immédiate d'après la construction de notre mini-langage de programmation.

c.q.f.d. $\Diamond\Diamond\Diamond$

Définition 2.3.2 *Soit un langage $L \subseteq (\Sigma \setminus \{B, \$\})^*$ et une MT M.*

- M décide *L si, pour tout $w \in (\Sigma \setminus \{B\})^*$, il existe un (unique) calcul de M de la forme $\gamma_0 \vdash \cdots \gamma_i \vdash \cdots$ partant de $\gamma_0 = (\epsilon, q_0, \$w)$ et, terminant en $\gamma_n = (w_1, \textbf{\textit{accept}}, w_2)$ si $w \in L$ et $\gamma_n = (w_1, \textbf{\textit{reject}}, w_2)$ si $w \notin L$, ceci pour certains n, w_1, et w_2.*
- M accepte *L si, pour tout mot $w \in (\Sigma \setminus \{B\})^*$, ou bien $w \in L$ et il existe un calcul de M partant de γ_0 et s'arrêtant sur une configuration **accept**, ou bien $w \notin L$ et soit M s'arrête en échec, soit M ne s'arrête pas sur w.*
- *L est* récursif *s'il existe une MT qui décide L.*
- *L est* récursivement énumérable *s'il existe une MT qui accepte L.*
- *L est* co-récursivement énumérable *si son complémentaire est récursivement énumérable.*

On notera $L(M)$ l'ensemble des mots acceptés par une machine M.
Remarques

1. L récursif ssi la fonction caractéristique de L est calculable
2. Si L est récursif, alors L est récursivement énumérable.

Théorème 2.3.2 *Il existe des langages non récursivement énumérables*

Preuve
On peut utiliser un argument de cardinalité. Mais aussi un langage explicite : w_i est une énumération des mots, M_i une énumération des machines de Turing (ces ensembles étant dénombrables, on admet ici connue une numérotation), $L = \{w_i \mid w_i \notin L(M_i)\}$ n'est pas r.e. En effet, s'il existait une machine M_L telle que, $w \in L$ ssi M_L s'arrête sur w avec succès, alors soit $M_i = M_L$:

$$w_i \notin L(M_i) \Leftrightarrow w_i \in L \Leftrightarrow w_i \in L(M_i)$$

Ce qui est absurde.

c.q.f.d. $\Diamond\Diamond\Diamond$

Proposition 2.3.3 *Si $f : (\Sigma \setminus \{B, \$\})^* \to (\Sigma \setminus \{B, \$\})^*$ est calculable, alors :*

1. *Pour toute fonction g calculable, $g \circ f$ est calculable.*

2. *Si L est récursif, alors $f^{-1}(L)$ est récursif.*
3. *Si L est récursivement énumérable, alors $f^{-1}(L)$ est récursivement énumérable.*

Preuve
Soient $M_f = (Q_f, q_{0,f}, \Sigma, \delta_f, \{B, \$\})$ une machine qui calcule f et $M_g = (Q_g, q_{0,g}, \Sigma, \delta_g, \{B, \$\})$ qui calcule g (resp. décide L, resp. accepte L). On exhibe une machine $M_{g\circ f}$ qui calcule $g \circ f$ comme suit :

- $Q_{g\circ f} \stackrel{\text{def}}{=} Q_g \uplus Q_f \uplus \{q_+\}$ contient tous les états de M_f, de M_g, ainsi qu'un état supplémentaire q_+ ;
- l'état initial $q_{0,g\circ f} \stackrel{\text{def}}{=} q_{0,f}$ est hérité de M_f ;
- la fonction de transition $\delta_{g\circ f}$ combine δ_f et δ_g comme suit :
  - si $q \in Q_f$ et $\delta_f(q,a) = (q', b, d)$ avec $q' \in Q_f$, alors $\delta_{g\circ f}(q,a) \stackrel{\text{def}}{=} (q', b, d)$;
  - si $q \in Q_f$ et $\delta_f(q,a) = (q', b, d)$ avec $q' \in \{\textbf{accept}, \textbf{reject}\}$, alors $\delta_{g\circ f}(q,a) \stackrel{\text{def}}{=} (q_+, b, d)$;
  - si $a \neq \$$, alors $\delta_{g\circ f}(q_+, a) \stackrel{\text{def}}{=} (q_+, a, \leftarrow)$;
  - si $\delta_g(q_{0,g}, \$) = (q, b, d)$, alors $\delta(q_+, \$) \stackrel{\text{def}}{=} (q, b, d)$;
  - si $q \in Q_g$ et $\delta_g(q,a) = (q', b, d)$, alors $\delta_{g\circ f}(q,a) \stackrel{\text{def}}{=} (q', b, d)$.

Si $w \in (\Sigma \setminus \{B, \$\})^*$, alors, par hypothèse, M_f s'arrête après n_w étapes et $M_f(w) = f(w)$. La machine $M_{g\circ f}$ arrive après n_w étapes dans une configuration (w_1, q_i, w_2) telle que $w_1 \cdot w_2 = f(w)$. Après au plus $|M_f(w)|$ étapes, la machine $M_{g\circ f}$ est dans la configuration initiale de M_g sur $f(w)$.

c.q.f.d. ◊◊◊

Théorème 2.3.4 *Un langage est récursif ssi il est récursivement énumérable et co-récursivement énumérable.*

Preuve
Si un langage est récursif, alors son complémentaire l'est aussi. Un sens de l'implication est donc immédiat. Si maintenant L est récursivement énumérable et co-récursivement énumérable, soient M_L et $\overline{M_L}$ les machines qui acceptent respectivement L et $\overline{L}$. On construit alors une machine M à deux rubans, qui commence par recopier sa donnée sur le second ruban, puis effectue alternativement un mouvement de M_L sur le premier ruban et un mouvement de $\overline{M_L}$ sur le second ruban. Dès que l'une des machines va entrer dans **accept**, la machine M s'arrête : en **accept** si c'est M_L qui a accepté, et en **reject** si c'est $\overline{M_L}$ qui a accepté. On définit symétriquement les transitions lorsque l'une des machines M_L ou $\overline{M_L}$ est sur le point de rejeter.

Pour toute entrée w, ou bien $w \in L$ et M_L s'arrête en **accept** (et $\overline{M_L}$ ou bien ne s'arrête pas ou bien s'arrête en **reject**). Dans ce cas, M s'arrête, ou bien quand M va accepter ou bien quand $\overline{M_L}$ va rejeter. Dans les deux cas M accepte w.

Ou bien $w \notin L$ et dans ce cas $\overline{M_L}$ s'arrête en **accept** sur w (et M_L pu bien ne s'arrête pas, ou bien s'arrête en **reject**). Dans ce cas, comme ci-dessus, M s'arrête en **reject**.

Donc, sur toute entrée w, M s'arrête. De plus $w \in L$ ssi M accepte w.

c.q.f.d. ◊◊◊

Définition 2.3.3 *Le* temps de calcul *d'une machine de Turing à k rubans sur la donnée w est le nombre de mouvements de la machine de Turing depuis la configuration initiale $(\epsilon, q_0, \$w)$ jusqu'à l'arrêt de la machine. On dit que « M calcule en temps $f(n)$ » si, pour toute donnée w de taille (nombre de symboles) inférieure ou égale à n, le temps de calcul de M sur w est inférieur ou égal à $f(n)$.*

*L'*espace de calcul *d'une machine de Turing I/O sur la donnée w est la longueur maximale d'un mot (privé de ses blancs finaux) inscrit sur les rubans de travail au cours du calcul de la machine sur w. On dit que « M calcule en espace $f(n)$ » si, pour toute donnée w de taille inférieure ou égale à n, l'espace de calcul de M sur w est inférieur ou égal à $f(n)$.*

On peut alors définir les classes de complexité :

Définition 2.3.4 *Si le langage L (resp. la fonction f) est décidé (resp. calculée) par une machine de Turing à k rubans en temps $g(n)$, on dit que $L \in \textbf{\textit{Time}}(g(n))$ (resp. $f \in \textbf{\textit{Time}}(g(n))$).*

Si le langage L (resp. la fonction f) est décidé (resp. calculée) par une machine de Turing I/O en espace $g(n)$ on dit que $L \in \textbf{\textit{Space}}(g(n))$ (resp. $f \in \textbf{\textit{Space}}(g(n))$).

2.4 Equivalence de types de machines

Il y en a beaucoup. Par exemple :

Théorème 2.4.1 *Pour toute machine de Turing M, on peut construire une machine de Turing M' qui n'a que deux états et telle que $L(M) = L(M')$*

Preuve
L'idée est la suivante : on suppose les états de M totalement ordonnés : soient $q_1, \ldots, q_n$ ces états.
À une configuration $a_1 \cdots a_n, q_i, a_{n+1} \cdots a_m$ de M correspond une configuration
$(q_0, a_1, \lhd) \cdots (q_0, a_n, \lhd) Q_0(q_i, a_{n+1}, \alpha)(q_0, a_{n+2}, \rhd) \cdots$ où $\alpha \in \{\lhd_d, \rhd_d, \lhd_i, \rhd_i\}$.
Une transition $q_i, a \mapsto q_j, a', \rightarrow$ correspond par exemple aux transitions :

$$\begin{array}{rcll}
Q_0, (q_i, a, \alpha) & \mapsto & Q_1, (q_{j-1}, a', \lhd_d), \rightarrow & \\
Q_1, (q_k, b, \rhd) & \mapsto & Q_1(q_{k+1}, b, \rhd_i), \leftarrow & \\
Q_1, (q_k, b, \rhd_i) & \mapsto & Q_1(q_{k+1}, b, \rhd_i), \leftarrow & \\
Q_1, (q_k, b, \lhd_d) & \mapsto & Q_1(q_{k-1}, b, \lhd_d), \rightarrow & \text{Si } k \geq 1 \\
Q_1, (q_0, b, \lhd_d) & \mapsto & Q_0, (q_0, b, \lhd), \rightarrow &
\end{array}$$

Une transition $q_i, a \mapsto q_j, a', \leftarrow$ correspond aux transitions :

$$\begin{array}{rcll}
Q_0, (q_i, a, \alpha) & \mapsto & Q_1, (q_{j-1}, a', \rhd_d), \leftarrow & \\
Q_1, (q_k, b, \lhd) & \mapsto & Q_1(q_{k+1}, b, \lhd_i), \rightarrow & \\
Q_1, (q_k, b, \lhd_i) & \mapsto & Q_1(q_{k+1}, b, \lhd_i), \rightarrow & \\
Q_1, (q_k, b, \rhd_d) & \mapsto & Q_1(q_{k-1}, b, \rhd_d), \leftarrow & \text{Si } k \geq 1 \\
Q_1, (q_0, b, \rhd_d) & \mapsto & Q_0, (q_0, b, \rhd), \leftarrow &
\end{array}$$

Expliquons comment fonctionne la machine lorsqu'elle va par exemple à droite. Elle remplace sur le caractère courant q_i par q_{j-1} puis par une suite d'aller-retour « décrémente » q_j sur le caractère courant et « incrémente » q_0 sur le caractère suivant. Elle termine lorsque le caractère courant porte q_0. Voici un exemple de déroulement avec la transition $q_5, a \mapsto q_2, a', \rightarrow$:
$Q_0(q_5, a, \alpha)(q_0, b, \rhd)$
$(q_1, a', \lhd_d) Q_1(q_0, b, \rhd)$
$Q_1(q_1, a', \lhd_d)(q_1, b, \rhd_i)$
$(q_0, a', \lhd_d) Q_1(q_1, b, \rhd_i)$
$Q_1(q_0, a', \lhd_d)(q_2, b, \rhd_i)$
$(q_0, a', \lhd) Q_0(q_2, b, \rhd_i)$

Il faut en plus une phase d'initialisation (qui n'utilise que Q_0) et considérer les blancs comme $(q_0, B, \rhd)$ dans les transitions ci-dessus.

Au total on utilisera un alphabet $\Sigma' = \Sigma \uplus Q \uplus \{q_0\} \times \Sigma \times \{\lhd, \rhd, \lhd_d, \rhd_d, \lhd_i, \rhd_i\}$.

c.q.f.d. ◊◊◊

Théorème 2.4.2 *Si M est une machine de Turing à k rubans calculant en temps $f(n) \geq n$, alors il existe une machine de Turing M' à 1 ruban, calculant en temps $O(f(n)^2)$ et telle que $M(x) = M'(x)$ pour tout x.*

Preuve
Voir le TD

c.q.f.d. ◊◊◊

Corollaire 2.4.3 *Les fonctions calculables (resp. les langages r.é.) pour une machine à k rubans sont les mêmes que pour une machine à un ruban.*

2.5 Machine de Turing universelle et problème de l'arrêt

On peut construire une MT *universelle* U qui, étant donné $<M, w>$ exécute M sur w : $U(<M, w>) = M(w)$. Le codage $<M, w>$ est défini par :

- $<M, w> = <M>;<w>$
- $<(Q, q_0, \Sigma, \delta, \{B, \$\})> = <Q>d_\Sigma<\Sigma>d_\delta<\delta>f_\delta$
- $<Q> = \overbrace{0\cdots0}^{1+\lfloor\log_2|Q|\rfloor}$: il y a autant de zéros que dans l'écriture en binaire du nombre d'éléments de Q. On suppose les états codés par des entiers $1, \ldots, |Q|$ et $q_0 = 1$, avec des 0 devant de façon à avoir tous même longueur.
- $<\Sigma> = \overbrace{0\cdots0}^{\lfloor\log_2|\Sigma|\rfloor}$: on suppose que B et \$ correspondent aux entiers 1 et 10 en binaire. Tous les symboles de Σ sont représentés par des entiers en binaire, complétés le cas échéant par des 0 à gauche, de manière à ce qu'ils aient tous même longueur.
- $<\delta>$ est codé comme suit : c'est une séquence de règles écrites sous la forme $c(q), c(a) \mapsto c(q'), c(b), m$; où le codage des états et symboles est par leur représentation binaire
- $<aw> = c(a)<w>$.

Le codage $<M, w>$ peut aussi n'utiliser qu'un alphabet de deux symboles : il suffit de recoder l'ensemble des symboles utilisés en binaire, ce que nous ne faisons pas pour des raisons de lisibilité.

Théorème 2.5.1 *Le langage* universel $L_U = \{<M, w> \mid w \in L(M)\}$ *est récursivement énumérable.*

Preuve
On utilise deux rubans sans perdre de généralité d'après le théorème 2.4.2.

On utilisera l'alphabet $\Sigma_U = \{0\ 1\ 0'\ 1'\ ;\ ,\ d_\Sigma\ d_\delta\ f_\delta\ \mapsto \rightarrow \downarrow \leftarrow\ B\ \$\}$. On peut sans difficulté ramener cet alphabet à 2 symboles (en plus de B et \$) comme vu en exercice.

La machine universelle commence par écrire la configuration initiale sur le deuxième ruban. Puis, si $\gamma \mapsto_M \gamma'$, alors on peut passer du codage de γ au codage de γ' (sur le deuxième ruban) par M'. Quand M accepte, M' aussi.

Pour la deuxième phase (simulation des mouvements de M), on procède comme suit :

1. Se placer au début de la première transition sur le premier ruban et au début de l'état sur le second ruban.
2. On progresse simultanément sur les deux rubans tant que les symboles sont identiques, tout en marquant les symboles du second ruban.
3. En cas d'échec (mismatch), on progresse jusqu'au début de la transition suivante sur le premier ruban et, sur le deuxième, on revient au début du code de l'état (en démarquant)
4. En cas de succès, on passe à la lecture des symboles, d'une part sur le premier ruban et d'autre part sur le second. En cas d'échec, on procède comme pour les échecs sur les états. En cas de succès (on arrive dans une phase de reconnaissance au symbole $\mapsto$), passer à l'étape suivante
5. Si le résultat de la transition de M est $\rightarrow, a', q'$, on remplace simplement $, q, a$ par a', q', (i.e. on se place juste avant le début de l'état puis on recopie la fin de la transition). Il faut ensuite remplacer éventuellement le symbole blanc qui suit l'état par son code.
6. Si le résultat de la transition de M est $\leftarrow, a', q'$, c'est un peu plus délicat, puisqu'il faut translater le symbole précédant $, q$, de $|<Q>| + 2$ vers la droite ; on utilise ici que les codes des états ont tous même longueur.

c.q.f.d. ◊◊◊

En termes moins techniques, une machine de Turing universelle est un interpréteur de machine de Turing et pourrait être obtenue en la programmant dans un langage quelconque puis « compilée » en machine de Turing.

Remarquons que les codages sont calculables. Par exemple :

Lemme 2.5.2 *La fonction qui à $<M>$ associe $<M,<M>>$ est calculable.*

Preuve
En effet, il suffit de recoder $<M>$ une deuxième fois en utilisant les codages des symboles 0 et 1.
Preuve

Théorème 2.5.3 *L_U n'est pas co-récursivement énumérable (et donc pas récursif).*

Preuve
$\overline{L_U} = \{<M,w> \mid w \notin L(M)\}$. Si ce langage était r.é, soit M_N une machine de Turing telle que $L(M_N) = \overline{L_U}$. On construit alors une machine de Turing D telle que $L(D) = \{<M> \mid <M> \notin L(M)\}$: sur la donnée $<M>$, D simplement recopie le code de M pour obtenir $<M,<M>>$ puis applique M_N. Mais alors,

$$<D> \in L(D) \Longleftrightarrow <D> \notin L(D)$$

Ce qui est absurde.

c.q.f.d. ◊◊◊

Le problème de l'arrêt dont l'intérêt pratique est évident est malheureusement indécidable.

Théorème 2.5.4 *$L_{arret} = \{<M,x> \mid M(x) \neq \perp\}$ est récursivement énumérable et pas co-récursivement énumérable (donc indécidable).*

Preuve
Tout d'abord, L_{arret} est récursivement énumérable : considérons la machine M_{arret} qui, étant donné $<M,x>$, simule la machine universelle et, quand celle-ci est sur le point d'accepter ou de rejeter, accepte. M_{arret} accepte L_{arret}.

Si L_{arret} était co-récursivement énumérable, soit $\overline{M_{arret}}$ une machine qui accepte $\overline{L_{arret}}$. On construit alors la machine H qui accepte $\{<M> \mid M(<M>) = \perp\}$ comme suit : H commence par dupliquer $<M>$, écrivant $<M,<M>>$, puis applique $\overline{M_{arret}}$. Si $\overline{M_{arret}}$ est sur le point de s'arrêter en **reject**, H entre dans un état spécial, où H se déplace indéfiniment vers la droite sans rien faire (et donc ne s'arrête pas).

$<H> \in L(H)$ si et seulement si $H(<H>) = \perp$. Mais, comme H ne rejette aucun mot, $H(x) = \perp$ si et seulement si $x \notin L(H)$. Donc $H(<H>) = \perp$ si et seulement si $<H> \notin L(H)$. Absurde. Il n'existe donc aucune machine de Turing qui accepte $\overline{L_{arret}}$.

c.q.f.d. ◊◊◊

Chapitre 3

Réductions : quelques problèmes indécidables

3.1 Réductions

D'une manière générale, on posera les problèmes de la façon suivante :

Donnée : $D \in S$

Question : Q

où S est un ensemble récursif et $Q \subseteq S$. Il s'agit d'une façon d'écrire l'ensemble $\{D \in S \mid Q\}$ où Q est cette fois vu comme un prédicat. On se passe le plus souvent de parler du codage de la donnée. Par exemple, on écrit

> **Donnée :** M, w où M est une machine de Turing et $w \in \{0,1\}^*$
>
> **Question :** M s'arrête-t-elle sur w ?
>
> est indécidable.

Cet énoncé est une façon de dire que $\{<M, w> \mid M(w) \neq \perp\}$ n'est pas récursif.

Pour montrer qu'un problème, donné sous cette forme, est indécidable, on procède souvent par *réduction* d'un problème connu pour être indécidable : Si P_1 est le problème

Donnée : $D_1 \in S_1$

Question : Q_1

est indécidable, et P_2 est le problème

Donnée : $D_2 \in S_2$

Question : Q_2

Pour prouver que P_2 est indécidable, il suffit d'exhiber une fonction récursive $f : S_1 \to S_2$ telle que, pour toute donnée $D_1 \in S_1$, $D_1 \in Q_1 \iff f(D_1) \in Q_2$. Voici un premier exemple.

Proposition 3.1.1 (L'arrêt universel) *Le problème suivant est indécidable :*

Donnée *: une machine de Turing M*

Question *: M s'arrête-t-elle sur toutes les données ?*

Preuve

On réduit le problème de l'arrêt : on considère la fonction f qui à $<M, x>$ associe $<M_x>$ où M_x est la machine qui écrit x puis simule M sur x (sans tenir compte de l'entrée). f est une fonction calculable : il suffit en effet d'ajouter $|x|+2$ états à M, qui permettent d'effacer le ruban et d'écrire x. De plus, M_x s'arrête sur toute entrée si et seulement si M s'arrête sur x.

c.q.f.d. $\Diamond\Diamond\Diamond$

3.2 Théorème de Rice

Théorème 3.2.1 *Si $\mathcal{P}$ est une propriété non triviale des langages récursivement énumérables, alors $\mathcal{P}$ est indécidable.*

Nous donnons ci-dessus l'énoncé «classique», mais en voici une version plus précise :

Pour tout ensemble $\mathcal{P}$ de (codes de) machines de Turing tel que :

1. pour toutes machines M, M' si $<M> \in \mathcal{P}$ et $L(M) = L(M')$ alors $<M'> \in \mathcal{P}$
2. on peut exhiber des machines M_1, M_2 telles que $<M_1> \in \mathcal{P}$ et $<M_2> \notin \mathcal{P}$

alors le problème suivant est indécidable :

Donnée : $<M>$

Question : $<M> \in \mathcal{P}$?

Preuve

Pour prouver ce résultat, nous utiliserons une réduction du problème de l'arrêt, un type de réduction qui sera très fréquemment utilisé par la suite.

Soit $\mathcal{P}$ un sous-ensemble non-trivial des langages récursivement énumérables. $\mathcal{P}$ est récursif ssi $\overline{\mathcal{P}}$ est récursif. Notons $M_\emptyset$ une machine dont le langage est vide.

En passant au complémentaire si nécessaire, on peut supposer sans perte de généralité que $<M_\emptyset> \notin \mathcal{P}$. D'après nos hypothèses, il existe une machine M_L t.q $<M_L> \in \mathcal{P}$. On note $L = L(M_L) \neq \emptyset$.

On réduit le problème de l'arrêt au problème de l'appartenance à $\mathcal{P}$ en construisant, à partir de la donnée $<M, x>$ du problème de l'arrêt, une donnée $<M_{<M,x>}>$ du problème de l'appartenance à $\mathcal{P}$, telle que $<M_{<M,x>}> \in \mathcal{P}$ ssi M s'arrête sur x.

Etant donnée une entrée y, $M_{<M,x>}$ simule l'exécution de M sur x puis en cas de succès simule l'exécution de M_L sur y. On observe que $L(M_{<M,x>}) \in \{L, \emptyset\}$ et que $<M_{<M,x>}> \in \mathcal{P}$ ssi M s'arrête sur x.

c.q.f.d. $\Diamond\Diamond\Diamond$

Les applications du théorème de Rice sont nombreuses. En voici quelques unes :

- Soit un mot w fixé, la question $w \in L(M)$ est indécidable.
  En particulier la question $\varepsilon \in L(M)$ est indécidable.
- La question $L(M) = \emptyset$ est indécidable.
- La question $L(M)$ est fini est indécidable.
- La question $L(M)$ est un langage régulier est indécidable.

Montrons à l'aide du théorème de Rice, que la question «M s'arrête-t-elle sur le mot vide ?» est indécidable. Si c'était le cas alors on saurait décider le problème si $\varepsilon \in L(M)$. En effet, on répond d'abord à la question «M s'arrête-t-elle sur le mot vide ?». Si M ne s'arrête pas, alors $\varepsilon \notin L(M)$. Sinon on exécute la machine M sur le mot vide pour avoir la réponse.

3.3 Problème de pavages

Le problème de pavage de $\mathbb{N} \times \mathbb{N}$ (les définitions sont semblables pour d'autres sous-ensembles de $\mathbb{Z} \times \mathbb{Z}$) est défini par :

Donnée : un ensemble fini $T = \{t_0, \ldots, t_k\}$ de *tuiles* dont t_0 est une tuile distinguée et deux sous-ensembles H, V de $T \times T$. (Les relations de compatibilité horizontale et verticale).

Question : existe-t-il une fonction de pavage $f : \mathbb{N} \times \mathbb{N} \mapsto T$ telle que $f(1,1) = t_0$ et, pour tout $i, j \in \mathbb{N}$, $(f(i,j), f(i+1,j)) \in H$, $(f(i,j), f(i,j+1)) \in V$.

Théorème 3.3.1 *Le problème de savoir, étant donnés T, t_0, V, H si $\mathbb{N} \times \mathbb{N}$ est pavable, est indécidable.*

Preuve
On réduit le problème de l'arrêt d'une machine sur le mot vide. Soit M une machine.

On choisit pour $T = (\Sigma \cup Q)^4 \cap \Sigma^*(Q + \epsilon)\Sigma^*$ (autrement dit, les mots de longueur 4 contenant au plus un symbole de Q).

$$H = \{(atc, tcb) \mid atc, tcb \in T, a, c, b, t \in \Sigma \cup Q, c = B \Rightarrow b = B\}$$

$$\begin{array}{rlllll}
V = & \{ & (\alpha qa\beta, & \alpha a'q'\beta) & \mid \alpha\beta \in \Sigma,\ a \in \Sigma,\ q \in Q,\ \delta(q,a) = (q', a', \rightarrow) & \} \\
\cup & \{ & (\alpha q, & \alpha a') & \mid \alpha \in \Sigma^2,\ a' \in \Sigma,\ q \in Q,\ \delta(q,a) = (q', a', \rightarrow) & \} \\
\cup & \{ & (a\alpha, & q'\alpha) & \mid \alpha \in \Sigma^2,\ q \in Q,\ \delta(q,a) = (q', a', \rightarrow) & \} \\
\cup & \{ & (bqa, & qba') & \mid a, b \in \Sigma,\ q \in Q,\ \delta(q,a) = (q', a', \leftarrow) & \} \\
\cup & \{ & (qa\beta, & ba'\beta) & \mid \beta \in \Sigma,\ a, b \in \Sigma,\ q \in Q,\ \delta(q,a) = (q', a', \leftarrow) & \} \\
\cup & \{ & (\beta a, & \beta q') & \mid \beta \in \Sigma^2,\ a \in \Sigma,\ \delta(q,a) = (q', a', \leftarrow) & \} \\
\cup & \{ & (\alpha qa\beta, & \alpha q'a'\beta) & \mid \alpha\beta \in \Sigma,\ a \in \Sigma,\ q \in Q,\ \delta(q,a) = (q', a', \downarrow) & \} \\
\cup & \{ & (\alpha q, & \alpha q') & \mid \alpha \in \Sigma^2,\ q \in Q,\ \delta(q,a) = (q', a', \downarrow) & \} \\
\cup & \{ & (\alpha, & \alpha) & \mid \alpha \in \Sigma^3 & \}
\end{array}$$

Enfin, on demande que le pavage satisfasse $f(1,1) = q_0\$B$.

Un pavage des n premières lignes horizontales du quart de plan supérieur correspond alors à une succession de n configurations de la machine de Turing, à partir de la configuration initiale. Plus précisément, si l'on ne retient que la première lettre de chaque tuile de la ligne n, on obtient la $n^{\text{ème}}$ configuration de M.

On montre ce résultat par récurrence sur n :

- pour $n = 1$, la première ligne contient la configuration initiale ; $f(1,1) = q_0\$B$ et donc $f(1,2) = \$BB$ et pour $m \geq 3$, $f(1,m) = BBB$.
- Si la suite des premières lettres des tuiles de la ligne n est une configuration $\gamma = w_1 q a w_2$ de la machine de Turing, alors la suite des première lettres des tuiles de la ligne $n+1$ est la configuration suivante de la machine : soit m la position de q dans γ. Supposons d'abord que $m > 1$. Pour $m - 1 \geq p \geq \max(1, m-2)$, $f(n,p) = w_1' bqaw_2'$ où $w_1' b$ est le suffixe de longueur $m - p$ de w_1 et w_2' est le préfixe de longueur $p + 2 - m$ de w_2, éventuellement complété de blancs.
  $f(n+1,p)$ est alors donné par :
  - $w_1' qba' w_2'$ si $\delta(q,a) = (q', a', \leftarrow)$,
  - $w_1' bqa' w_2'$ si $\delta(q,a) = (q', a', \downarrow)$,
  - $w_1' ba' q' w_2'$ si $\delta(q,a) = (q', a', \rightarrow)$.

  On conclut, grâce aux compatibilités verticales. Par exemple, $p \geq m + 2$, alors $f(n,p) \in \Sigma^4$ et $f(n+1,p) = f(n,p)$. En effet, les seules tuiles verticales compatibles sont (α, α) et $(\alpha b, \alpha q')$ (avec $\delta(q,a) = (q', a', \leftarrow)$). Mais on montre que ce dernier cas n'est pas possible en considérant $f(n, p+1)$: comme $p > m + 1$, $f(n, p+1) \in \Sigma^4$ et donc, par compatibilité verticale, $f(n+1, p+1) \in \Sigma^4 \cup Q \cdot \Sigma^3$. Si l'on avait $f(n+1,p) = \alpha q'$, on aurait une tuile horizontale de la forme $(\alpha q', \beta q'')$ ou $(\alpha q', \beta)$, ce qui n'est pas le cas.
  De même, pour $p < m - 3$, $f(n+1,p) = f(n,p)$. Restent les cas $p = m-3$, $p = m$, $p = m+1$ pour lesquels on montre par compatibilités horizontale et verticale que l'on obtient les lettres voulues.
  Enfin, si $m = 1$, le seul mouvement possible est à droite et la seule tuile verticale possible impose $f(n+1,1) = \$q'w'$ si $\delta(q, \$) = (q', \$, \rightarrow)$. Puis, pour $p > 2$, $f(n+1,p) = f(n,p)$ comme ci-dessus et $f(n+1,2) = q'w'b$ par compatibilités verticale et horizontale.

c.q.f.d. $\Diamond\Diamond\Diamond$

Nous énonçons sans preuve (car elle est très difficile) l'indécidabilité du problème sans tuile distinguée.

Théorème 3.3.2 *Le problème de savoir, étant donnés T, V, H si $\mathbb{Z}\times\mathbb{Z}$ est pavable, est indécidable.*

Preuve
La preuve peut être consultée dans [3] (appendix A).

c.q.f.d. ◊◊◊

A partir de ce résultat, on peut aussi déduire un résultat sur $\mathbb{N} \times \mathbb{N}$. Au préalable, nous introduisons le lemme de König qu'on retrouve dans de nombreuses preuves.

Lemme 3.3.3 (Lemme de Koenig) *Soit A un arbre de degré fini (i.e. tout noeud admet un nombre fini de successeurs) et comportant un nombre de noeuds infini ; alors A admet une branche infinie.*

Preuve
Nous exhibons la branche infinie de la manière suivante. Partant de la racine, on choisit l'un des successeurs de celle-ci dont le sous-arbre a un nombre infini de noeuds. Il en existe au moins un car le nombre de successeurs est fini. En itérant ce procédé au niveau de chaque nouveau sous-arbre choisi, on construit la branche infinie.

c.q.f.d. ◊◊◊

Théorème 3.3.4 *Le problème de savoir, étant donnés T, V, H si $\mathbb{N}\times\mathbb{N}$ est pavable, est indécidable.*

Preuve
Nous affirmons que l'existence d'un pavage dans $\mathbb{N} \times \mathbb{N}$ est équivalent à l'existence d'un pavage dans $\mathbb{Z} \times \mathbb{Z}$. Si on dispose d'un pavage dans $\mathbb{Z} \times \mathbb{Z}$ alors par troncature on obtient un pavage de $\mathbb{N} \times \mathbb{N}$.

Supposons qu'on dispose d'un pavage de $\mathbb{N} \times \mathbb{N}$. On construit un arbre de pavages finis. La racine est l'unique pavage de la surface vide. A une hauteur i, se trouvent les différents pavages du carré $[-i, i] \times [-i, i]$. Un pavage de $[-i-1, i+1] \times [-i-1, i+1]$ a pour père l'unique pavage de $[-i, i] \times [-i, i]$ dont il est l'extension. En raison de l'existence du pavage de $\mathbb{N}\times\mathbb{N}$, il y a des noeuds à toutes les hauteurs. Le lemme de König assure l'existence d'une séquence infinie de pavages $\{P_i\}_{i\in\mathbb{N}}$. $\bigcup_{i\in\mathbb{N}} P_i$ est le pavage recherché.

c.q.f.d. ◊◊◊

3.4 Problème de correspondance de Post

Le problème de correspondance de Post (PCP) :

Donnée : Deux suites de mots finies $(u_1, \ldots, u_n)$ et $(v_1, \ldots, v_n)$ de même longueur

Question : existe-t-il un entier $k > 0$ et une suite d'indices $i_1, \ldots, i_k$ tels que

$$u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$$

Par exemple, soient $(u_i, v_i)_{i=1,\ldots,4}$ donnés par

i	1	2	3	4
u_i	a	b	ca	abc
v_i	ab	ca	a	c

Cette instance de PCP a une solution (12314).
Une variante de ce problème est le PCP contraint :

Donnée : Deux suites de mots finies $(u_1, \ldots, u_n)$ et $(v_1, \ldots, v_n)$ de même longueur

Question : existe-t-il un entier $k > 0$ et une suite d'indices $\mathbf{i_1 = 1}, \ldots, i_k$ tels que

$$u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$$

Théorème 3.4.1 *Le PCP contraint est indécidable.*

Preuve
On réduit le problème de l'arrêt sur le mot vide. Soit M une machine de Turing. On note q_f le seul état d'arrêt de M. Les paires de mots de PCP contraint sont :

	mots v_i (dans l'ordre)	mots u_i correspondant	
1.	$\lhd$	$\lhd q_0\$\star$	
2.	a	a	pour $a \in \Sigma$
3.	qa	$a'q'$	Si $\delta(q,a) = q', a', \rightarrow$
4.	aqb	$q'ab'$	si $\delta(q,b) = q', b', \leftarrow$
5.	qa	$q'a'$	si $\delta(q,a) = q', a', \downarrow$
6.	$q\star$	$aq'\star$	si $\delta(q,B) = q', a', \rightarrow$
7.	$bq\star$	$q'ba'\star$	si $\delta(q,B) = q', a', \leftarrow$
8.	$q\star$	$q'a\star$	si $\delta(q,B) = q', a', \downarrow$
9.	$\star$	$\star$	
10.	$q_f a$	aq_f	
11.	$q_f\star$	$q_e\star$	
12.	$aq_e\star$	$q_e\star$	
13.	$\$q_e \star \rhd$	$\rhd$	

Expliquons le principe de la réduction. A l'aide des correspondances de 1 à 10 on ne peut fabriquer (sans blocage ultérieur) qu'une paire de phrases $u = u_{i_1} \dots u_{i_n}$ et $v = v_{i_1} \dots v_{i_n}$
Montrons que PCP contraint a une solution ssi M s'arrête sur le mot vide :

Si M s'arrête sur le mot vide Soit $s_1 = q_0\$ \vdash \cdots \vdash w_k q_f w'_k = s_k$ le calcul de M sur le mot vide (les blancs en fin de ruban n'apparaissent pas dans les configurations). Pour tout $i < k$, s_i, s_{i+1} sont de l'une des formes suivantes :
- $s_i = t_i q a w_i$ et $s_{i+1} = t_i a' q' w_i$ avec $\delta(q,a) = q', a', \rightarrow$
- $s_i = t_i a q b w_i$ et $s_{i+1} = t_i q' a b' w_i$ avec $\delta(q,b) = q', b', \leftarrow$
- $s_i = t_i q a w_i$ et $s_{i+1} = t_i q' a' w_i$ avec $\delta(q,a) = q', a', \downarrow$
- L'un des cas ci-dessus en retirant la partie à droite de q (tête de lecture à droite du ruban)...

Dans chaque cas, on remarque que $s_i\star$ peut s'écrire comme la concaténation $v_{m_1} \cdots v_{m_i}$ et $s_{i+1}\star$ s'écrit comme la concaténation $u_{m_1} \cdots u_{m_i}$. Par exemple, dans le premier cas : On utilise $|t_i|$ fois la correspondance 2, puis une fois la correspondance 3, puis $|w_i|$ fois la correspondance 2, puis la correspondance 9.

On obtient alors une solution de PCP contraint comme suit :

1	$2^*3\dots$	$\cdots$							
$\lhd q_0\$w\star$	$s_1\star$	$\cdots$	$s_k\star$	$w_k a q_f w'_k\star$	$\cdots$	$w_n q_e\star$	$w_{n-1} q_e\star$	$\cdots$	$\rhd$
$\lhd$	$s_0\star$	$\cdots$	$s_{k-1}\star$	$w_k q_f a w'_k\star$	$\cdots$	$w_n q_f\star$	$w_{n-1} a q_e\star$	$\cdots$	$\$q_e \star \rhd$

Réciproquement, si PCP contraint a une solution Alors montrons que M s'arrête sur le mot vide. Soit $u_{i_1} \cdots u_{i_m} = v_{i_1} \cdots v_{i_m}$. On montre, par récurrence sur k qu'il existe un p, et des mots t, t' tels que $u_{i_1} \cdots u_{i_k} = \lhd s_1 \star \cdots s_p \star t$, $v_{i_1} \cdots v_{i_k} = \lhd s_1 \star \cdots s_{p-1} \star t'$ et ou bien s_p est une configuration finale de la machine, ou bien
- t' est un préfixe de s_p et t est un préfixe de s_{p+1}
- Si t' ne contient pas de symbole d'état et ne se termine pas par $\star$, alors $t' = t$, sinon $t' \vdash_M t$.

Pour $k = 1$, $u_{i_1} = u_1 = \lhd q_0\$w\star$ et $v_{i_1} = v_1 = \lhd$. On a bien, pour $p = 1$, $u_{i_1} = \lhd s_1\star$, $t = t' = \epsilon$.

Si la propriété est vraie pour k, considérons la paire suivante $u_{i_{k+1}}, v_{i_{k+1}}$. Si s_p était déjà une configuration finale, il n'y a rien à faire. Sinon, $s_p = t' \cdot t'_p$ et $s_{p+1} = t \cdot t_p$. De plus, comme on a une solution de PCP contraint, $v_{i_{k+1}}$ est un préfixe de $t'_p \star t u_{i_{k+1}}$. Considérons successivement tous les $v_{i_{k+1}}$ possibles :

- 1 : impossible car $\lhd$ n'apparaît pas dans $t'_p \star t u_{i_{k+1}}$
- 2 : dans ce cas, $t'_p = v_{i_{k+1}} t''_p$ et on a la propriété voulue
- 3, 4, 5 : $t'_p = v_{i_{k+1}} t''_p$, t' ne contient pas de symbole d'état, et donc $t = t'$ et $t' \cdot v_{i_{k+1}} \vdash_M t \cdot u_{i_{k+1}}$
- 6, 7, 8 : comme t'_p ne contient pas $\star$, $t'_p \star = v_{i_{k+1}}$. Donc $s_p \star = t' \cdot v_{i_{k+1}}$. De plus, $t = t'$ et $t \cdot u_{i_{k+1}} = s_{p+1} \star$. Il suffit alors de choisir des mots vides pour les nouveaux t, t'
- 9 : t'_p est vide et il suffit de choisir des mots vides pour les nouveaux t, t'
- 10, 11 : l'état final a été atteint.
- 12 : impossible pour des raisons identiques à 9.
- 13 : t' doit être vide puisque $v_{i_{k+1}}$ est un préfixe de $t'_p \star t u_{i_{k+1}}$. De plus, on doit avoir $t'_p = \$q_e$, ce qui n'est pas possible. Ce cas n'a pas lieu

c.q.f.d. $\Diamond\Diamond\Diamond$

Théorème 3.4.2 *Le PCP est indécidable.*

Preuve
On réduit PCP contraint à PCP comme suit, en supposant (sans perte de généralité) qu'il n'y a pas de paire (ϵ, ϵ).

Si $(u_1, \ldots, u_n)$, $(v_1, \ldots, v_n)$ est une instance de PCP contraint, on considère un alphabet augmenté des lettres $\bullet, \rhd, \lhd$ et l'instance de PCP : $(\lhd\overline{u_1}, \overline{u_1}, \ldots, \overline{u_n}, \bullet\rhd), (\lhd\bullet\widetilde{v_1}, \widetilde{v_1}, \ldots, \widetilde{v_n}, \rhd)$ où $\overline{\epsilon} = \widetilde{\epsilon} = \epsilon$ et $\overline{a \cdot w} = \bullet a \overline{w}$ et $\widetilde{a \cdot w} = a \bullet \widetilde{w}$.

Si PCP contraint a une solution $u_{i_1} \cdots u_{i_m} = v_{i_1} \cdots v_{i_m}$, alors $\lhd\overline{u_{i_1}} \cdots \overline{u_{i_m}}\bullet\rhd = \lhd\bullet\widetilde{v_{i_1}} \cdots \widetilde{v_{i_m}}\rhd$ est une solution de PCP.

Réciproquement, si PCP admet une solution, notons $(u'_0, \ldots, u'_n, u'_{n+1})$ et $(v'_0, \ldots, v'_{n+1})$ l'instance du problème : il existe une suite d'indices telle que $u'_{i_1} \cdots u'_{i_m} = v'_{i_1} \cdots v'_{i_m}$. Notons que, pour tout i, $u'_i = \epsilon$ ou bien u'_i commence par $\bullet$, ou bien $i = 1$. Si $u'_{i_1} = \epsilon$, alors soit k le plus petit indice tel que $u'_{i_k} \neq \epsilon$. Par hypothèse et par construction, $v'_{i_1} \neq \epsilon$ et sa première lettre est $a \notin \{\bullet, \lhd\}$. À l'inverse, la première lettre de u'_{i_k} est dans $\{\bullet, \lhd\}$. Ce qui est absurde. Il en résulte que $u'_{i_1} \neq \epsilon$. Dans ce cas, la première lettre de u'_{i_1} est dans $\{\bullet, \lhd\}$ et donc aussi la première lettre du premier v_{i_k} non vide. Ce n'est possible que si $i_1 = 1$ et cette première lettre est $\lhd$.

Soit maintenant ϕ le morphisme défini sur $(\Sigma \cup \{\bullet, \lhd, \rhd\})^*$ par $\phi(\bullet) = \phi(\lhd) = \phi(\rhd) = \epsilon$ et $\phi(a) = a$ sinon. On montre, par récurrence sur k que $\phi(u'_{i_1} \cdots u'_{i_k}) = u_1 \cdot u_{i_2} \cdots u_{i_k}$ et $\phi(v_{i_1} \cdots v_{i_k}) = v_1 \cdot v_{i_2} \cdots v_{i_k}$, si $1 \leq k < m$. Il en résulte que, si PCP a une solution, alors PCP contraint aussi, en prenant l'image par ϕ.

c.q.f.d. $\Diamond\Diamond\Diamond$

Si tous les mots apparaissant dans les paires sont différents de ε alors on peut supprimer le symbole $\lhd$ de la première paire. C'est le cas du PCP contraint obtenu par la réduction du problème de l'arrêt. Par contre, il est indispensable si cette contrainte n'est pas vérifiée ainsi que le montre l'exemple suivant.

$$u_1 = b, v_1 = ab, u_2 = a, v_2 = \varepsilon$$

Il n'existe pas de solution au probleme contraint et le PCP obtenu admet la solution suivante : $u = (\bullet a)(\bullet b)(\bullet\rhd)$ et $v = (\varepsilon)(\bullet a \bullet b \bullet)(\rhd)$

Chapitre 4

Fonctions récursives

4.1 Introduction

L'objectif de cette partie est d'examiner un autre modèle de calcul que les machines de Turing et de montrer qu'il a le même pouvoir expressif que les machines de Turing (donc d'apporter un témoin à la thèse de Church). L'intérêt de ce modèle est double. D'une part il est fonctionnel et correspond à une autre approche de la programmation. D'autre part il est structurel (fonctions de base et constructeurs) ; ceci permet à la fois des preuves par induction sur la structure et d'autre part d'introduire des sous-classes de fonctions à terminaison garantie.

4.2 Fonctions récursives primitives

f désignera une fonction de $\mathbb{N}^k$ dans $\mathbb{N}$. Les fonctions totales sont aussi des *applications*. $f(n_1, \ldots, n_k) = \perp$ si f n'est pas définie en $n_1, \ldots, n_k$.

Les *fonctions initiales* sont

- les projections $P_k^i : \mathbb{N}^k \mapsto \mathbb{N}$. $P_k^i(n_1, \ldots, n_k) = n_i$;
- le successeur $S(n) = n + 1$;
- la fonction nulle $Z(n) = 0$;
- la fonction constante 0 (à 0 arguments).

Un ensemble d'applications F est *fermé par composition* si, pour toutes fonctions $\xi : \mathbb{N}^m \mapsto \mathbb{N}$ et $\psi_1, \ldots, \psi_m : \mathbb{N}^n \mapsto \mathbb{N}$, la fonction $\phi : \mathbb{N}^n \mapsto \mathbb{N}$ définie par $\phi(\vec{n}) = \xi(\psi_1(\vec{n}), \ldots, \psi_m(\vec{n}))$ est dans F. On note $\mathsf{Comp}_m(\xi, \psi_1, \ldots, \psi_m)$ la fonction ϕ ainsi obtenue.

Un ensemble d'applications F est *fermé par récursion primitive* si, pour toutes fonctions $\xi, \psi \in F$, la fonction ϕ définie par récurrence par :

$$\phi(m, \vec{n}) = \begin{cases} \xi(\vec{n}) & \text{si } m = 0, \\ \psi(\phi(m-1, \vec{n}), m-1, \vec{n}) & \text{si } m > 0 \end{cases}$$

est dans F. On note $\mathsf{Prim}(\xi, \psi)$ la fonction ϕ ainsi obtenue.

Définition 4.2.1 *L'ensemble des* fonctions récursives primitives *est le plus petit ensemble contenant les fonctions initiales, clos par composition et récursion primitive.*

Toutes les fonctions récursives primitives s'obtiennent ainsi à partir de fonctions de base et des opérations Prim et Comp_n.

Exemple 4.2.1 *Montrons que les opérations arithmétiques sont primitives récursives.*

- *La décrémentation dans les entiers naturels est primitive récursive :*
  $n - - = Si\ n = 0\ Alors\ 0\ Sinon\ n - 1\ Finsi$

- *La soustraction dans les entiers naturels est primitive récursive :*
  $m - n = Si\ n = 0\ Alors\ m\ Sinon\ (m - (n-1)) - -\ Finsi$
- *L'addition « $f(n,m) \stackrel{\text{def}}{=} n + m$ » est primitive récursive :* $f = \mathsf{Prim}(P_1^1, \mathsf{Comp}_1(S, P_3^1))$. *Ce qui peut s'écrire de manière plus intuitive :*
  $n + m = Si\ n = 0\ Alors\ m\ Sinon\ ((n-1) + m) + 1\ Finsi$
- *La multiplication « $g(n,m) = n \times m$ » est primitive récursive :* $g = \mathsf{Prim}(Z, \mathsf{Comp}_2(+, P_3^1, P_3^3))$
  Ce qui peut s'écrire de manière plus intuitive :
  $n \times m = Si\ n = 0\ Alors\ 0\ Sinon\ ((n-1) \times m) + m\ Finsi$
- *Le « test à zéro » est primitif récursif :*
  $n? = Si\ n = 0\ Alors\ 0\ Sinon\ 1\ Finsi$

Nous développons d'autres constructions récursives primitives utiles comme la somme bornée :

$$g(x_1, \ldots, x_n, y) = \sum_{t \leq y} f(x_1, \ldots, x_n, t)$$

En effet,
$g(x_1, \ldots, x_n, y) =$
$Si\ y = 0\ Alors\ f(x_1, \ldots, x_n, 0)$
$Sinon\ g(x_1, \ldots, x_n, y-1) + f(x_1, \ldots, x_n, y)\ Finsi$

Soit $\mathcal{R}$ une relation n-aire, on note $\chi_{\mathcal{R}}$ la fonction indicatrice n-aire indicatrice de $\mathcal{R}$. On dit que $\mathcal{R}$ est récursive primitive si sa fonction indicatrice l'est. A titre d'exercice, le lecteur peut démontrer que $x = y$, $x < y$, etc. sont des relations primitives récursives. Ainsi des fonctions définies par :
$h(x_1, \ldots, x_n) = Si\ \mathcal{R}(x_1, \ldots, x_n)\ Alors\ f(x_1, \ldots, x_n) Sinon\ g(x_1, \ldots, x_n)\ Finsi$
sont bien récursives primitives. En effet :
$h(x_1, \ldots, x_n) = \chi_{\mathcal{R}}(x_1, \ldots, x_n) f(x_1, \ldots, x_n) + (1 - \chi_{\mathcal{R}}(x_1, \ldots, x_n)) g(x_1, \ldots, x_n)$

Une autre construction primitive récursive est le schéma existentiel borné noté :
$f(x_1, \ldots, x_n, z) = \exists t \leq z\ \mathcal{R}(x_1, \ldots, x_n, t)$
et défini par $f(x_1, \ldots, x_n, z) = 1$ si $\exists t \leq z$ qui vérifie $\mathcal{R}(x_1, \ldots, x_n, t)$ et 0 sinon.
Ce schéma est bien récursif primitif car $f(x_1, \ldots, x_n, z)$ est la fonction caractéristique de $(\sum_{y \leq z} \chi_{\mathcal{R}}(x_1, \ldots, x_n, y) \geq 1)$

Une autre construction primitive récursive est le schéma de minimisation bornée noté :
$f(x_1, \ldots, x_n, z) = \mu t \leq z\ \mathcal{R}(x_1, \ldots, x_n, t)$
et défini par $f(x_1, \ldots, x_n, z) = t$ si t est le plus petit entier compris entre 0 et z qui vérifie $\mathcal{R}(x_1, \ldots, x_n, t)$ et 0 sinon.
Ce schéma est bien récursif primitif car il s'écrit :
$f(x_1, \ldots, x_n, z) =$
$Si\ z = 0\ Alors\ 0$
$Sinon\ f(x_1, \ldots, x_n, z-1)(\sum_{y \leq z-1} \chi_{\mathcal{R}}(x_1, \ldots, x_n, y) \geq 1)$
$+z(\sum_{y \leq z-1} \chi_{\mathcal{R}}(x_1, \ldots, x_n, y) = 0)(\chi_{\mathcal{R}}(x_1, \ldots, x_n, z) = 1)\ Finsi$

Une application simple de ce schéma est la division : $x/y = \mu t \leq x\ y \times (t+1) > x$

4.2.1 De la programmation fonctionnelle à la programmation impérative ($LFOR$)

Nous allons montrer que les fonctions récursives primitives ont leur « pendant » en programmation impérative. Nous considérons $LFOR$ un langage de programmation sur les entiers composé des instructions élémentaires (copie, incrémentation, décrémentation) et des structures de contrôle if et for et la concaténation. Remarquons que ce langage de programmation a l'agréable propriété que tout programme se termine (donc ne nécessite pas de test d'arrêt). Afin de comparer ce langage

de programmation aux fonctions récursives primitives, nous supposons que le programme dispose d'une instruction *return* x_i où x_i est une variable du programme.

Fin d'arriver au résultat principal de ce paragraphe, nous introduisons un codage récursif primitif d'une suite de k entiers $x_1, \ldots, x_k$ par un entier que nous noterons $\langle x_1, \ldots, x_k \rangle$. Le codage d'un entier est l'identité. Le codage de deux entiers $\langle x_1, x_2 \rangle$ est défini par :

$$\langle x_1, x_2 \rangle = (x_1 + x_2)(x_1 + x_2 + 1)/2 + x_1$$

qui consiste à énumérer les paires par diagonales $x_1 + x_2$ croissantes en parcourant les diagonales par x_1 croissant.

Puis le codage de k entiers est obtenu par la formule itérative :

$$\langle x_1, \ldots, x_k \rangle = \langle x_1, \langle x_2, \ldots, x_k \rangle \rangle$$

Ce codage a la propriété évidente qu'il est supérieur ou égal à chacune des composantes codées. On peut par conséquent appliquer les schémas de minimisation et existentiel borné pour extraire les composantes.

$$\pi_k^i(z) = \mu x_i \leq z \; \exists x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_k \leq z \; z = \langle x_1, \ldots, x_k \rangle$$

Nous affirmons que :

Théorème 4.2.2 *Les fonctions calculées par les programmes LFOR sont exactement les fonctions primitives récursives.*

Preuve
Nous laissons le soin au lecteur de montrer que toute fonction primitive récursive est calculable par un programme $LFOR$ et nous nous consacrons à la démonstration de la réciproque.

Nous allons montrer comment transformer la boucle for suivante. Les autres traductions sont immédiates.

$for\ x_{n+1}\ from\ 1\ to\ x_1\ do$
$\quad x_2 := f_2(x_1, \ldots, x_{n+1})$
$\quad \ldots$
$\quad x_n := f_n(x_1, \ldots, x_{n+1})$
$endfor$

Les fonctions manipuleront des entiers codant des suites de valeurs correspondant aux variables du programme. Nous introduisons les fonctions $\overline{f_i}$.

$$\overline{f_i}(\sigma, x) = f_i(\pi_n^1(\sigma), \ldots, \pi_n^n(\sigma), x)$$

Puis la fonction f qui représente l'effet d'un tour de boucle sur les variables $x_1, \ldots, x_n$ lorsque la variable x_{n+1} est égale à x. La séquence σ est le codage de la suite des variables $x_1, \ldots, x_n$.

$$f(\sigma, x) = \langle \pi_n^1(\sigma), \overline{f_2}(\sigma, x), \ldots, \overline{f_n}(\sigma, x) \rangle$$

La fonction g traduit l'effet de x tours de boucle sur les variables $x_1, \ldots, x_n$.

$$g(\sigma, x) = si\ x = 0\ alors\ \sigma sinon\ f(g(\sigma, x-1), x)\ finsi$$

La fonction h traduit l'effet du programme sur toutes les variables.

$$h(\sigma) = \langle \pi_{n+1}^1(\sigma), \pi_n^2(g(\langle \pi_{n+1}^1(\sigma), \ldots, \pi_{n+1}^n(\sigma) \rangle, \pi_{n+1}^1(\sigma))), \ldots, \pi_n^n(g(\langle \pi_{n+1}^1(\sigma), \ldots, \pi_{n+1}^n(\sigma) \rangle, \pi_{n+1}^1(\sigma))), \pi_{n+1}^1(\sigma) \rangle$$

c.q.f.d. $\Diamond\Diamond\Diamond$

On généralise le codage de séquences de taille fixe au codage de séquence de taille variable. En codant la longueur comme premier élément de la suite. Autrement dit, le codage de $x_1, \ldots, x_m$ est $\langle m, x_1, \ldots, x_m \rangle$. La projection $\pi(i, \sigma)$ s'écrit alors à l'aide du langage $LFOR$ (utilisation possible d'apès le théorème 4.2.2) :

$x := \pi_2^1(\sigma)$
$if\ x < i\ then\ return\ 0$
$for\ y\ from\ 1\ to\ i\ do$
    $\sigma := \pi_2^2(\sigma)$
$endfor$
$if\ x = i\ then\ return\ \sigma\ else\ return\ \pi_2^1(\sigma)$

La longueur d'une séquence est aussi primitive récursive ($\pi_2^1(\sigma)$). Le programme suivant permet d'écrire une valeur v à la position i de la séquence σ. Nous noterons la fonction associée $w(i, v, \sigma)$. Nous avons légèrement triché en utilisant un for « décroissant ». Nous laissons au lecteur le soin de l'implémenter par un for « croissant ».

$x := \pi_2^1(\sigma)$
$if\ x < i\ then\ return\ \sigma$
$for\ y\ from\ 1\ to\ i\ do$
    $\sigma' := \pi_2^2(\sigma')$
$endfor$
$if\ x = i\ then\ \sigma' := v\ else\ \sigma' := \langle v, \pi_2^2(\sigma') \rangle$
$for\ y\ from\ i\ downto\ 1\ do$
    $\sigma' := \langle \pi(y, \sigma), \sigma' \rangle$
$endfor$
$return\ \langle x, \sigma' \rangle$

Enfin le programme ci-dessous concatène une valeur v à la séquence σ. Nous noterons la fonction associée $conc(v, \sigma)$.

$x := \pi_2^1(\sigma)$
$\sigma' := v$
$for\ y\ from\ x\ downto\ 1\ do$
    $\sigma' := \langle \pi(y, \sigma), \sigma' \rangle$
$endfor$
$return\ \langle x + 1, \sigma' \rangle$

4.2.2 Hiérarchie de Grzegorczyk

On définit par récurrence sur n la suite de fonctions primitives récursives comme suit :

$$\begin{array}{rcl} \psi_0(m) & = & m+1 \\ \psi_{n+1}(0) & = & \psi_n(1) \\ \psi_{n+1}(m+1) & = & \psi_n(\psi_{n+1}(m)) \end{array}$$

On montre par récurrence sur n qu'il s'agit bien d'une suite de fonctions récursives primitives.

Par récurrence sur m, on montre successivement les résultats suivants pour les premières fonctions de la hiérarchie :

$\psi_1(m) = m + 2$ car $\psi_1(m+1) = \psi_0(\psi_1(m)) = (m+2) + 1 = (m+1) + 2$

$\psi_2(m) = 2m + 3$ car $\psi_2(m+1) = \psi_1(\psi_2(m)) = (2m+3) + 2 = 2(m+1) + 3$

$\psi_3(m) = 2^{m+3} - 3$ car $\psi_3(m+1) = \psi_2(\psi_3(m)) = 2(2^{m+3} - 3) + 3 = 2^{(m+1)+3} - 3$

$$\psi_4(m) = \overset{m+3}{\left\{ \begin{array}{l} \\ \\ 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \end{array} \right.} -3$$

Lemme 4.2.3 (Propriétés de monotonie de ψ)

1. *pour tous* n, m, $\psi_n(m) > n + m$;
2. *les fonctions* ψ_n *sont (strictement) croissantes :* $\psi_n(m+1) > \psi_n(m)$;
3. *pour tous* n, m, $\psi_{n+1}(m) > \psi_n(m)$ *d'où pour tout* k, $\psi_{n+k}(m) \geq \psi_n(m) + k$.
4. *pour tous* n, m, k, $\psi_k^m(n) \leq \psi_{k+1}(n+m)$ *où* ψ_k^m *désigne la fonction* ψ_k *composée* m *fois.*

Preuve

1. On montre
$$\psi_n(m) > n + m \tag{4.2.3.a}$$
par induction sur n puis sur m. Le cas $n = 0$ se réduit à $\psi_0(m) = m+1 > m$. Pour $n > 0$, on considère d'abord le cas $m = 0$: $\psi_n(m) = \psi_n(0) = \psi_{n-1}(1) > n-1+1$ par hyp. ind., c.-à-d. $\psi_n(m) > n = n + m$. Pour $m > 0$, on a $\psi_n(m) = \psi_{n-1}(\psi_n(m-1))$, c.-à-d. $= \psi_{n-1}(x)$ pour $x = \psi_n(m-1) \geq n + m$ par hyp. ind. En utilisant encore une fois l'hypothèse d'induction, on déduit $\psi_{n-1}(x) \geq x + n$, soit $\psi_n(m) \geq 2n + m > n + m$ puisque $n > 0$.
2. On montre
$$\psi_n(m+1) > \psi_n(m), \tag{4.2.3.b}$$
ce qui équivaut à $\psi_n(m+k) \geq \psi_n(m) + k$, par induction sur n et en utilisant la question précédente. Pour $n = 0$, on a $\psi_n(m+1) = m+1+1 > m+1 = \psi_n(m)$. Supposons maintenant $n > 0$. Alors
$$\begin{aligned}\psi_n(m+1) &= \psi_{n-1}(\psi_n(m)) \\ &> n - 1 + \psi_n(m) \text{ par (4.2.3.a)} \\ &\geq \psi_n(m).\end{aligned}$$
3. On montre
$$\psi_{n+1}(m) > \psi_n(m), \tag{4.2.3.c}$$
ce qui équivaut à $\psi_{n+k}(m) \geq \psi_n(m) + k$, par induction sur m et en utilisant les questions précédentes. Pour $m = 0$, on a $\psi_{n+1}(m) = \psi_n(1) > \psi_n(0) = \psi_n(m)$ directement grâce à (4.2.3.b). Pour $m > 0$, on a $\psi_{n+1}(m) = \psi_n(\psi_{n+1}(m-1)) = \psi_n(x)$ pour $x = \psi_{n+1}(m-1)$, c.-à-d. pour $x > n + m$ par (4.2.3.a). On déduit $\psi_n(x) > \psi_n(m)$ par (4.2.3.b). Ainsi on a bien $\psi_{n+1}(m) > \psi_n(m)$.
4. On montre $\psi_k^m(n) \leq \psi_{k+1}(n+m)$ par récurrence sur m. Le cas $m = 0$ s'exprime : $n \leq \psi_{k+1}(n)$ et se déduit du fait que $\psi_{k+1}(n) \geq \psi_1(n) = n + 2$. Démontrons le pas de la récurrence. $\psi_k^{m+1}(n) \leq \psi_k(\psi_{k+1}(n+m)) = \psi_{k+1}(n+m+1)$ par hypothèse de récurrence et par croissance de ψ_k.

c.q.f.d. ◊◊◊

Lemme 4.2.4 *Pour tous* k, m, n, $\psi_k(\psi_m(n)) \leq \psi_{2+\max(k,m)}(n)$.

Preuve
On s'appuie sur le lemme 4.2.3. Soit $M = \max(k, m)$.
$$\begin{aligned}\psi_k(\psi_m(n)) &< \psi_M(\psi_{M+1}(n)) \text{ par la monotonie de } \psi \\ &= \psi_{M+1}(n+1) \\ &= \psi_{M+1}(\psi_1(n-1)) \text{ si } n > 0 \\ &\leq \psi_{M+1}(\psi_{M+2}(n-1)) \text{ encore par monotonie} \\ &= \psi_{M+2}(n).\end{aligned}$$
D'autre part, si $n = 0$, alors $\psi_{M+1}(n+1) = \psi_{M+2}(0)$ et on a encore l'inégalité.

c.q.f.d. ◊◊◊

Proposition 4.2.5 *Pour toute fonction récursive primitive à un argument ξ, il existe une fonction de la hiérarchie de Grzegorczyk ψ_n telle que*

$$\forall m\ \xi(m) \leq \psi_n(m)$$

Preuve
On prouve par récurrence sur le nombre d'opérations utilisées dans la construction des fonctions primitives récursives que, si ϕ est primitive récursive, alors il existe un k_ϕ tel que $\phi(\vec{n}) \leq \psi_{k_\phi}(\max \vec{n})$.

- C'est vrai pour les fonctions initiales : $P_k^i(\vec{n}) \leq \psi_0(\max \vec{n})$, $S(n) \leq \psi_0(n)$, $Z(n) \leq \psi_0(n)$.
- Si ϕ est obtenue par composition de $\xi, \phi_1, \ldots, \phi_m$, il suffit de choisir

  $$k_\phi = 2 + \max(k_\xi, k_{\phi_1}, \ldots, k_{\phi_n}),$$

  en utilisant le lemme 4.2.4.
- Si ϕ est obtenue par récursion primitive :

  $$\phi(m, \vec{n}) = \begin{cases} \xi(\vec{n}) & \text{si } m = 0 \\ \psi(\phi(m-1, \vec{n}), m-1, \vec{n}) & \text{si } m > 0 \end{cases}$$

  On montre par récurrence sur m que $\phi(m, \vec{n}) \leq \psi_{k_\psi}^m(\psi_{k_\xi}(\max(\vec{n})))$.
  Si $m = 0$, alors $\phi(0, \vec{n}) = \xi(\vec{n}) \leq \psi_{k_\xi}(\max(\vec{n}))$ par hypothèse de récurrence.
  Si $m > 0$, par hypothèse de récurrence,

  $$\phi(m-1, \vec{n}) \leq \psi_{k_\psi}^{m-1}(\psi_{k_\xi}(\max(\vec{n}))).$$

  Par croissance de ψ_{k_ψ} et comme $\psi_{k_\psi}^m(n) \geq n + m$, on a aussi

  $$\begin{array}{rcl} \phi(m, \vec{n}) & \leq & \psi_{k_\psi}(\max(\psi_{k_\psi}^{m-1}(\psi_{k_\xi}(\max(\vec{n}))), m-1, \vec{n})) \\ & \leq & \psi_{k_\psi}(\psi_{k_\psi}^{m-1}(\psi_{k_\xi}(\max(\vec{n})))) \end{array}$$

  Par ailleurs, $\psi_k^m(n) \leq \psi_{k+1}(n+m)$ (lemme 4.2.3), donc

  $$\phi(m, \vec{n}) \leq \psi_{1+k_\psi}(m + \psi_{k_\xi}(\max(\vec{n}))) \leq \psi_{1+k_\psi}(2\psi_{k_\xi}(\max(m, \vec{n})))$$

  $$\leq \psi_{1+k_\psi}(\psi_2(\psi_{k_\xi}(\max(m, \vec{n})))) \leq \psi_{\max(1+k_\psi, \max(2,k_\xi)+2)+2}(\max(m, \vec{n}))$$

c.q.f.d. ◊◊◊

4.2.3 La fonction d'Ackermann

La *fonction d'Ackermann* à deux arguments est la fonction $A(n, m) = \psi_n(m)$. La *fonction d'Ackermann* à un argument est la fonction $A(n) = \psi_n(n)$.

Théorème 4.2.6 *Les fonctions d'Ackermann à un et à deux arguments ne sont pas récursives primitives.*

Preuve
Si la fonction à deux arguments est récursive primitive alors la fonction à un argument est primitive récursive. Si la fonction à un argument est primitive récursive alors d'après la proposition 4.2.5, il existerait un entier k tel que, pour tout n, $A(n, n) \leq \psi_k(n)$. Mais en choisissant $n = k + 1$, on obtient $\psi_{k+1}(k+1) \leq \psi_k(k+1)$, ce qui contredit les propriétés de monotonie des fonctions ψ_m (proposition 4.2.3).

c.q.f.d. ◊◊◊

4.3 Fonctions récursives (partielles, totales)

Un ensemble de fonctions F est *fermé par minimisation* (resp. *fermé par minimisation totale*) si, pour toute fonction (resp. pour toute application $\xi \in F$ telle que $\forall \vec{n}\ \exists m\ \xi(\vec{n}, m) = 0 \wedge \forall k < m\ \xi(\vec{n}, k)$), la fonction ϕ définie par

$$\phi(\vec{n}) = \min\{m \mid (\xi(\vec{n}, m) = 0) \wedge \forall k < m\ \xi(\vec{n}, k) \neq \perp\}$$

est dans F.

La composition de fonctions totales est étendue aux fonctions partielles : la composée n'est définie que lorsque les fonctions composantes le sont. De même, la récursion primitive appliquée à des fonctions partielles : $\phi = \mathsf{Prim}(\xi, \psi)$ est définie en $(m, \vec{n})$ si $m = 0$ et $\xi(\vec{n}) \neq \perp$ ou bien $m > 0$, $\phi(m-1, \vec{n}) \neq \perp$ et $\psi(\phi(m-1, \vec{n}), m-1, \vec{n}) \neq \perp$.

Définition 4.3.1 *L'ensemble des* fonctions récursives partielles (resp. totales) *est le plus petit ensemble de fonctions qui contient les fonctions initiales et qui est fermé par composition, récursion primitive et minimisation (resp. totale).*

Nous affirmons que le modèle des fonctions récursives est équivalent au modèle des machines de Turing. Plus précisément :

Théorème 4.3.1 *Les fonctions (resp. applications) calculées par les programmes $LWHILE$ sont exactement les fonctions récursives partielles (resp. totales).*

Preuve
Nous laissons le soin au lecteur de montrer comment une fonction (resp. une application) récursive peut être calculée par un programme du langage $LWHILE$ (resp. qui se termine) et nous nous concentrons sur la réciproque. D'après nos résultats sur le codage, il nous suffit de montrer comment simuler le programme suivant :
while $f(x) \neq 0$ *do*
    $x := g(x)$
endwhile
return x
où f et g sont des fonctions récursives et f est à valeurs dans $\{0, 1\}$.

Nous définissons :
$h(x, n) = if\ x = 0\ then\ x\ else\ g(h(x, n-1))$
Cette fonction calcule l'effet de n tours de boucle.

Puis nous définissons :
$tour(x, n) = min(\{n \mid f(h(x, n)) = 0 \wedge \forall n' < n\ f(h(x, n)) \neq \perp\})$
Cette fonction renvoie (s'il existe le tour de sortie).

La fonction recherchée est alors $h(x, tour(x))$.

Il est immédiat que la minimisation est totale ssi le programme contenant la boucle *while* se termine pour toutes les valeurs de x.

c.q.f.d. ◊◊◊

Théorème 4.3.2 *La fonction d'Ackermann est récursive totale.*

Preuve
Nous écrivons un programme $LWHILE$ qui calcule $A(m, n)$.
while $\pi_2^1(\sigma) < 2m + 1 \vee \pi(2m+1, \sigma) \neq n$ *do*
    $\sigma := w(1, \pi(1, \sigma) + 1, \sigma)$
    $\sigma := w(2, \pi(1, \sigma) + 1, \sigma)$
    for x *from* 2 *to* $m + 1$ *do*

$if\ \pi_2^1(\sigma) = 2(x-1) \wedge \pi(2x-3,\sigma) = 1\ then$
$\sigma := conc(0,\sigma)$
$\sigma := conc(\pi(2x-2,\sigma),\sigma)$
$elseif\ \pi_2^1(\sigma) \geq 2x \wedge \pi(2x-3,\sigma) = \pi(2x,\sigma)\ then$
$\sigma := w(2x-1,\pi(2x-1,\sigma)+1,\sigma)$
$\sigma := w(2x,\pi(2x-2,\sigma),\sigma)$
$endif$
$endfor$
$endwhile$
$return\ \pi(2m+2,\sigma)$

Ce programme maintient dans σ une suite de $2k$ valeurs $v_1,\ldots,v_{2k}$ avec $1 \leq k \leq m+1$ qui vérifient les assertions suivantes (au début de chaque tour de boucle) :

- Pour tout $i \leq k$, $v_{2i} = A(i-1,v_{2i-1})$. Autrement dit, la suite de valeurs peut être écrite $v_1, A(0,v_1), v_3, A(1,v_3),\ldots,v_{2i-1}, A(i-1,2i-1)$.
- Pour tout $i < k$, $v_{2i-1} < v_{2i+2}$. Autrement dit la valeur courante de v_{2i-1} est strictement inférieure à $A(i,v_{2i+1})$.
- Si $k < m+1$ alors $v_{2k+1} = 0$. Autrement dit on n'a pas encore le moyen de calculer $A(k,0)$.

Initialement la suite de valeurs est égale à $0, 1 = A(0,0)$. A chaque tour de boucle, le programme calcule la paire suivante $x, A(0,x)$ et si la valeur x correspond à la valeur $A(1,y)$ stockée dans la paire suivante $y, A(1,y)$, le programme a le moyen de calculer la paire $y+1, A(1,y+1) = A(0,A(1,y))$. L'extension de la suite correspond au cas où $v_{2k-1} = 1$ ce qui permet de calculer la paire $0, A(k,0) = v_{2k} = A(k-1,1)$.

Supposons que ce programme ne se termine pas. Alors on démontre par récurrence sur i que les valeurs v_{2i+1} croissent strictement par pas de 1 (avec entre chaque incrément des tours où la valeur est stationnaire) en démarrant à 0. Par conséquent v_{2m+1} doit atteindre n. Ce qui contredit la non terminaison.

c.q.f.d. ◊◊◊

Théorème 4.3.3 (Kleene) *Toute fonction récursive partielle s'obtient à l'aide d'une seule minimisation d'une fonction primitive récursive.*

Preuve
Soit f une fonction récursive partielle, f est calculée par une machine de Turing disons $\mathcal{M}$. A l'aide de notre construction primitive récursive de séquence, on peut coder par un entier une exécution de $\mathcal{M}$. Par exemple on peut choisir $\langle conf_1, conf_2, \ldots, conf_n\rangle$ avec $conf_i$ un entier codant une configuration par $\langle c(\$), c(a_1),\ldots,c(a_{i-1}),c(q),c(a_i),\ldots,c(a_m)\rangle$. Ici c associe à chaque caractère et état un entier distinct.

Il est facile d'écrire un programme $LFOR$ qui teste si un entier code une exécution dont le dernier état est terminal. Le programme recherché est un programme $LWHILE$ comportant une boucle *while* externe qui énumère les entiers et exécute le programme $LFOR$ pour tester l'entier. La condition d'arrêt du programme est un test positif. La traduction de la boucle *while* vue plus haut fournit l'expression recherchée.

c.q.f.d. ◊◊◊

Chapitre 5

A la frontière de la décidabilité

5.1 Introduction

L'objet de ce dernier chapitre est de montrer deux exemples de problèmes où un changement apparemment mineur de l'énoncé transforme le statut du problème (de décidable à indécidable et vice versa). Nous avons choisi de prendre des exemples associés à des problèmes sur des modèles étendant les automates et en particulier des problèmes d'accessibilité qui sont importants à la fois d'un point de vue théorique mais aussi pratique.

5.2 Automates à une file [4]

Un automate à une file est un automate doté d'une file contenant un mot w. Les opérations de l'automate consiste à « envoyer » un message a sur la file, i.e. à tranformer le mot en wa ou à « recevoir » un message a sur la file à condition que a soit la première lettre du mot, i.e. à transformer $w = aw'$ en w'. On note $!a$ la première opération et $?a$ la deuxième opération. $IO(\Sigma)$ désigne l'ensemble des opérations sur l'alphabet Σ : $IO(\Sigma) = \{*a \mid * \in \{!, ?\} \wedge a \in \Sigma\}$.

Définition 5.2.1 *Un automate à une file $\mathcal{A} = (\Sigma, Q, q_0, F, \rightarrow)$ avec :*
- *Σ, un alphabet fini*
- *Q, un ensemble fini d'états incluant q_0 l'état initial et F l'ensemble des états terminaux.*
- *X, un ensemble fini d'horloges.*
- *$\rightarrow \subseteq Q \times IO(\Sigma) \times Q$, la relation de transition. On note $(q, io, q') \in \rightarrow$ par $q \xrightarrow{io} q'$*

Une configuration d'un automate à une file est une paire (q, w) avec $q \in Q$ et $w \in \Sigma^*$.

Définition 5.2.2 *Une exécution d'un automate à une file est une séquence :*

$$(q_0, w_0) \xrightarrow{io_1} (q_1, w_1) \dots \xrightarrow{io_n} (q_n, w_n)$$

avec $w_0 = \varepsilon$ le mot vide, $\forall 1 \leq i \leq n$ $q_{i-1} \xrightarrow{io_i,} q_i$ et :
- *Si $io_i = !a$ alors $w_i = aw_{i-1}$;*
- *Si $io_i = ?a$ alors $w_{i-1} = aw_i$.*

On définit le problème de l'accessibilité dans un automate à une file ainsi. Existe-t-il une exécution qui rencontre un état de F ?

Théorème 5.2.1 *Le problème de l'accessibilité dans les automates à une file est indécidable.*

Preuve
Nous allons réduire le problème de l'arrêt d'une machine de Turing $\mathcal{M}$ sur le mot vide à ce problème. Pour cela l'automate associé $\mathcal{A}$ simule l'exécution de la machine de Turing ainsi :

- L'alphabet de $\mathcal{A}$ est l'alphabet de $\mathcal{M}$ enrichi d'un marqueur $\sharp$.
- Les états de $\mathcal{A}$ (notés $Q_{\mathcal{A}}$) sont soit des états de $\mathcal{M}$ (notés $Q_{\mathcal{M}}$) soit des états intermédiaires (notés Q'). Une configuration (q, w) est dite *configuration de simulation* si $q \in Q_{\mathcal{M}}$ et w représente un état de $\mathcal{M}$ (i.e. $w = w_1\sharp w_2\sharp$ où le premier $\sharp$ marque la position de la tête de lecture et le deuxième $\sharp$ marque la fin de la configuration). Les configurations (q, w) telles que $q \in Q'$ sont appelées configurations intermédiaires.
- Une exécution de $\mathcal{A}$ consiste en une suite de séquences d'exécution précédée d'une séquence initiale consistant à écrire sur la file $\$\sharp B\sharp$ et à se positionner sur l'état initial de $\mathcal{M}$. Chaque séquence conduit d'une configuration de simulation à la configuration de simulation suivante (vis à vis de $\mathcal{M}$) à travers des configurations intermédiaires.
- La simulation d'un pas de $\mathcal{M}$ procède ainsi :
  1. $\mathcal{A}$ maintient en mémoire q et les trois dernières lettres lues de w, disons xyz. Si y est différent de $\sharp$ alors $\mathcal{A}$ écrit x et lit une nouvelle lettre t, mémorisant yzt.
  2. Si y est égal à $\sharp$ alors z est la lettre sous la tête de lecture. En fonction de $\delta(q, z) = (q', \$, d)$, $\mathcal{A}$ écrit les caractères $x'y'z'$ correspondant au nouvel état de la bande de $\mathcal{M}$ (avec éventuellement ajout d'un blanc).
  3. $\mathcal{A}$ recopie ensuite les caractères jusqu'au deuxième $\sharp$ ce qui, après passage à l'état q', conduit à la nouvelle configuration de simulation.

Par conséquent, $\mathcal{A}$ atteint **accept** ou **reject** ssi $\mathcal{M}$ s'arrête.

c.q.f.d. $\Diamond\Diamond\Diamond$

5.3 Automates à une file avec pertes [1, 5]

Un automate à une file avec pertes a la même syntaxe et les mêmes configurations qu'un automate à une file. Seule la notion de séquence d'exécution diffère.

Définition 5.3.1 *Une exécution d'un automate à une file avec pertes est une séquence :*

$$(q_0, w_0) \xrightarrow{io_1} (q_1, w_1) \dots \xrightarrow{io_n} (q_n, w_n)$$

avec $w_0 = \varepsilon$ le mot vide et $\forall 1 \leq i \leq n$:
- *Soit il existe $q_{i-1} \xrightarrow{!a} q_i$ et $w_i = aw_{i-1}$;*
- *Soit il existe $q_{i-1} \xrightarrow{?a} q_i$ et $w_{i-1} = aw_i$.*
- *Soit $q_i = q_{i-1}$, $w_{i-1} = w_1aw_2$ et $w_{i-1} = w_1w_2$*

Autrement dit, l'automate avec pertes peut perdre une lettre quelconque du mot de la file.

On définit la relation suivante entre mots $w \prec w'$ si notant $w = a_1 \dots a_n$, il existe une décomposition de w' : $w = w_0a_1w_1 \dots w_{n-1}a_nw_n$. Autrement dit, le mot w « se plonge » dans w'.

Lemme 5.3.1 (Higman) *Soit une suite infinie de mots $(w_0, w_1, \dots)$. Alors il existe deux indices $i < j$ tels que $w_i \prec w_j$.*

Preuve
Considérons l'ensemble $\mathcal{S}$ des suites infinies qui ne vérifient pas la propriété du lemme et supposons que $\mathcal{S} \neq \emptyset$.

On fabrique un élément $\mathbf{w} = (w_0, w_1, \dots)$ de $\mathcal{S}$ de la façon suivante.
- On choisit pour w_0 un mot de taille minimale parmi les premiers mots des éléments de $\mathcal{S}$.
- Une fois choisis $w_0, w_1, \dots, w_i$ on choisit pour w_{i+1} un mot de taille minimale parmi les w_{i+1} tels que $w_0, w_1, \dots, w_{i+1}$ constitue un préfixe fini d'un élément de $\mathcal{S}$. Par construction, un tel choix est toujours possible.

Par construction $\mathbf{w} \in \mathcal{S}$. Dans la suite $\mathbf{w} = (w_0, w_1, \ldots)$, une des lettres (disons a) apparaît infiniment souvent en première position des w_i. Soit $w_{\alpha(0)}, w_{\alpha(1)}, \ldots$ la sous-suite extraite avec $w_{\alpha(i)} = aw'_{\alpha(i)}$. Nous laissons le soin au lecteur de vérifier que $\mathbf{w}' = w_0, \ldots, w_{\alpha(0)-1}, w'_{\alpha(0)}, w'_{\alpha(1)}, \ldots$ appartient à $\mathcal{S}$. D'autre part, $|w'_{\alpha(0)}| < |w_{\alpha(0)}|$ contredisant la définition de $\mathbf{w}$. Par conséquent $\mathcal{S}$ est vide ce qui établit le lemme.

c.q.f.d. ◊◊◊

Nous voulons établir la décidabilité de l'accessibilité pour les automates à une file avec pertes. Nous étendons la relation $\prec$ sur les configurations par $(q, w) \prec (q'w')$ si $q = q'$ et $w \prec w'$. De manière évidente la propriété du lemme de Higman est vérifiée pour les configurations : soit une suite infinie de configurations $((q_0, w_0), (q_1, w_1), \ldots)$. Alors il existe deux indices $i < j$ tels que $(q_i, w_i) \prec (q_j, w_j)$. En effet, sachant qu'au moins un état apparaît infiniment souvent, il suffit de considérer la suite extraite correspondante.

Pour savoir si une configuration (q, w) est accessible depuis une configuration (q', w'), nous allons construire une représentation finie de $BAcc(q, w)$ l'ensemble des configurations à partir desquelles (q, w) est accessible. Cette représentation finie doit permettre de tester l'appartenance d'une configuration à cet ensemble.

Pour un ensemble de configurations E, notons $E^{\uparrow}$ l'ensemble des configurations supérieures ou égales à une configuration de E. Les ensembles clos supérieurement sont définis par la propriété $E = E^{\uparrow}$. Remarquons que $BAcc(q', w')$ est clos supérieurement. En effet soit (q', w') qui permet d'atteindre (q, w) et soit (q', w'') avec $w' \prec w''$, alors en perdant les messages « en excès » de w'', on atteint la configuration (q', w').

Pour obtenir une représentation finie d'un ensemble clos supérieurement, il suffit de prendre les éléments minimaux de cet ensemble. En effet, puisqu'ils sont tous incomparables, ils ne peuvent être en nombre infini car de nouveau cela contredirait le lemme de Higman. Appelons $Min(E)$ l'ensemble des éléments minimaux de E. Puisque E est clos supérieurement, $E = Min(E)^{\uparrow}$ ce qui montre qu'un ensemble clos supérieurement est déterminé de manière univoque par ses élements minimaux. De plus le test d'appartenance est effectif car il suffit de comparer la configuration avec les éléments minimaux.

Le lemme suivant est la clé de la décidabilité de l'accessibilité.

Lemme 5.3.2 *Soit $\{E_n\}_{n\in\mathbb{N}}$ une suite d'ensembles de configurations supérieurement et croissants ; cette suite se stabilise.*

Preuve
Dans le cas contraire, chaque fois qu'un ensemble n'est pas égal au précédent, ceci signifie qu'il y a une configuration de cet ensemble qui n'est supérieure ou égale à aucune des configurations des ensembles précédents. Nous sélectionnons l'un de ces éléments. En itérant le procédé, on obtient une suite infinie de configurations qui ne sont pas supérieures ou égales aux configurations précédentes de la suite. Mais ceci contredit le lemme de Higman.

c.q.f.d. ◊◊◊

Théorème 5.3.3 *Le problème de l'accessibilité de (q, w) à partir de (q_0, w_0) dans les automates à une file avec pertes est décidable.*

Preuve

Remarquons que $BAcc(q, w) = \bigcup_{n\in\mathbb{N}} BAcc_n(q, w)$ avec $BAcc_n(q, w)$ l'ensemble des configurations à partir desquelles (q, w) est accessible en au plus n pas. Bien sûr $BAcc_0(q, w) = \{(q, w)\}$. Nous désirons obtenir $Bacc(q, w)$ en calculant par itérations successives $Min(E_n)$ pour des ensembles E_n croissants tels que $Bacc_n(q, w) \subseteq E_n \subseteq Bacc(q, w)$ et $E_0 = BAcc_0(q, w)$.

Supposons que nous ayons calculé $Min(E_n)$. Pour atteindre en un pas à partir d'un $(q'', w'') \notin E_n$ une configuration (q', w') supérieure ou égale à une configuration $(q', w_{min}) \in Min(E_n)$, il faut

effectuer soit une réception de message soit un envoi de message. La perte de message implique que $(q'', w'') \prec (q', w')$ et donc que $(q'', w'') \in E_n$.

- Soit $(q'', w'') \xrightarrow{!a} (q', w')$, alors $w' = w''a$. Par conséquent, si la dernière lettre de w_{min} est a, i.e. $w_{min} = w'_{min}a$ ceci est équivalent (moyennant des suppressions de messages avant le franchissement de la transition) à $w'_{min} \prec w''$. Sinon ceci est équivalent à $w_{min} \prec w''$.
- Soit $(q'', w'') \xrightarrow{?a} (q', w')$, alors $w'a = w''$. Moyennant des suppressions de messages avant le franchissement de la transition ceci est équivalent à $w_{min}a \prec w''$.

Pour résumer,

$$E_{n+1} = E_n \cup \bigcup_{(q',w') \in Min(E_n), q'' \xrightarrow{!a} q'} \{(q'', w'') \mid w' \prec w''\}$$
$$\cup \bigcup_{(q',w'a) \in Min(E_n), q'' \xrightarrow{!a} q'} \{(q'', w'') \mid w' \prec w''\}$$
$$\cup \bigcup_{(q',w') \in Min(E_n), q'' \xrightarrow{?a} q'} \{(q'', w'') \mid w'a \prec w''\}$$

Par conséquent, $Min(E_{n+1})$ est le sous-ensemble des éléments minimaux de l'ensemble fini ci-dessous :

$$Min(E_n) \cup \{(q'', w') \mid \exists (q', w') \in Min(E_n)\ \exists q'' \xrightarrow{!a} q'\}$$
$$\cup \{(q'', w') \mid \exists (q', w'a) \in Min(E_n)\ \exists q'' \xrightarrow{!a} q'\}$$
$$\cup \{(q'', w'a) \mid \exists (q', w') \in Min(E_n)\ \exists q'' \xrightarrow{?a} q'\}$$

L'algorithme consiste donc à calculer les $Min(E_n)$ jusqu'à stabilisation ce qui arrive certainement en raison du lemme précédent puis à tester si $(q', w') \prec (q_0, w_0)$ pour au moins un $(q', w') \in Min(\bigcup E_n)$.

c.q.f.d. ◊◊◊

5.4 Automates temporisés [2]

Un automate temporisé est un automate enrichi avec des horloges (notées $X = \{x_1, \ldots, x_n\}$). Ces horloges contraignent les transitions par des gardes. Une garde est une conjonction de gardes élémentaires. Une garde élementaire s'écrit $x_i \bowtie c$ où c est une constante entière est $\bowtie \in \{<, >, \leq, \geq, =, \neq\}$.

Une valuation d'horloge v est une application de X dans $\mathbb{R}_{\geq 0}$. v satisfait la garde $x_i \bowtie c$ ce qu'on note $v \vDash x_i \bowtie c$ ssi $v(x_i) \bowtie c$. Une garde est satisfaite si toutes ses gardes élémentaires sont satisfaites. On notera $\Gamma(X)$ l'ensemble des gardes sur X. De plus lors du franchissement d'une transition, un sous-ensemble d'horloges peut être remis à zéro. On note $v[X']$ la valuation définie par $\forall x \in X'\ v[X'](x) = 0$ et $\forall x \notin X'\ v[X'](x) = v(x)$.

Le comportement d'un automate temporisé peut être informellement décrit comme suit. Il laisse s'écouler du temps dans l'état courant, puis franchit (si possible) en temps nul une transition dont la garde est satisfaite. Autrement dit, il alterne des pas de temps (éventuellement nuls) et des pas discrets. Lors d'un pas de temps les horloges évoluent de façon synchrone. On note $v + d$, pour d réel positif, la valuation définie par $\forall x \in X\ v + d(x) = v(x) + d$.

Définition 5.4.1 *Un automate temporisé $\mathcal{A} = (\Sigma, Q, q_0, F, X, \rightarrow)$ avec :*

- *Σ, un alphabet fini*
- *Q, un ensemble fini d'états incluant q_0 l'état initial et F l'ensemble des états terminaux.*
- *X, un ensemble fini d'horloges.*
- *$\rightarrow \subseteq Q \times \Gamma(X) \times \Sigma \times 2^X \times Q$, la relation de transition. On note $(q, \gamma, a, R, q') \in \rightarrow$ par $q \xrightarrow{\gamma, a, R} q'$*

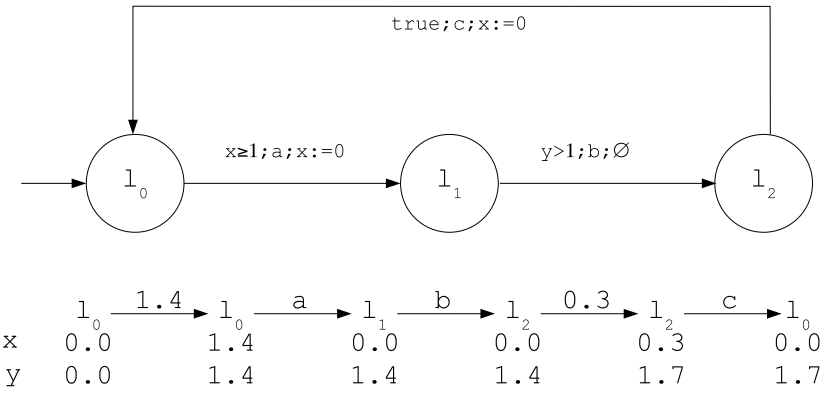


FIG. 5.1 – Un automate temporisé et une exécution

La sémantique d'exécution d'un automate temporisé est formalisée ci-dessous.

Définition 5.4.2 *Une exécution d'un automate temporisé est une séquence :*

$$(q_0, v_0) \xrightarrow{d_1} (q_0, v_0 + d_1) \xrightarrow{\gamma_1, a_1, R_1} (q_1, v_1) \xrightarrow{d_2} (q_1, v_1 + d_2) \dots \xrightarrow{\gamma_n, a_n, R_n} (q_n, v_n)$$

avec $\forall 1 \leq i \leq n\ q_{i-1} \xrightarrow{\gamma_i, a_i, R_i} q_i,\ v_{i-1} + d_i \vDash \gamma_i,\ v_i = (v_{i-1} + d_i)[R_i]$.
Dans ce cas, le mot temporisé $(a_1, d_1) \dots (a_n, d_n)$ *et le* mot non temporisé $a_1 \dots a_n$ *sont acceptés par l'automate temporisé.*

La figure 5.1 illustre les définitions précédentes. Le problème de l'accessibilité dans les automates temporisés consiste à savoir si étant donné un état q, il existe une configuration accessible (q, v) depuis (q_0, v_0).

Théorème 5.4.1 *Le problème de l'accessibilité d'un état q d'un automate temporisé est décidable.*

La preuve de ce théorème consiste à définir une relation d'équivalence $\sim$ entre configurations et un graphe des classes dont les arcs sont étiquetés soit par les transitions de l'automate soit le passage du temps (noté τ). Nous ne détaillons pas pour l'instant la relation et le graphe mais indiquons les propriétés qu'ils vérifient :

- $(q, v) \sim (q', v') \Rightarrow q = q'$. Aussi on notera (q, Z) une classe d'équivalence.
- Le nombre de classes d'équivalence est fini et $\{(q_0, v_0)\}$ est une classe d'équivalence. Le graphe des classes doit être calculable.
- Soit $(q, Z) \xrightarrow{\gamma, a, R} (q', Z')$ un arc « événementiel » du graphe des classes alors :
  $\forall (q, v) \in (q, Z)\ \exists (q', v') \in (q', Z')\ (q, v) \xrightarrow{\gamma, a, R} (q', v')$.
- Soit $(q, Z) \xrightarrow{\tau} (q, Z')$ un arc « temporisé » du graphe des classes alors :
  $\forall (q, v) \in (q, Z)\ \exists d \in \mathbb{R}_{\geq 0}\ (q, v + d) \in (q, Z')$.
- Soit (q, Z) une classe et $(q, v) \in (q, Z)$. Si $(q, v) \xrightarrow{\gamma, a, R} (q', v')$ alors :
  il existe un arc $(q, Z) \xrightarrow{\gamma, a, R} (q', Z')$ t.q. $(q', v') \in (q', Z')$.
- Soit (q, Z) une classe et $(q, v) \in (q, Z)$ alors pour tout $d \in \mathbb{R}_{\geq 0}$:
  il existe un arc $(q, Z) \xrightarrow{\tau} (q', Z')$ t.q. $(q, v + d) \in (q', Z')$.

La première condition assure que le problème de l'accessibilité dans le graphe des classes est décidable. Les deux conditions suivantes assurent (par récurrence sur la longueur du chemin) que

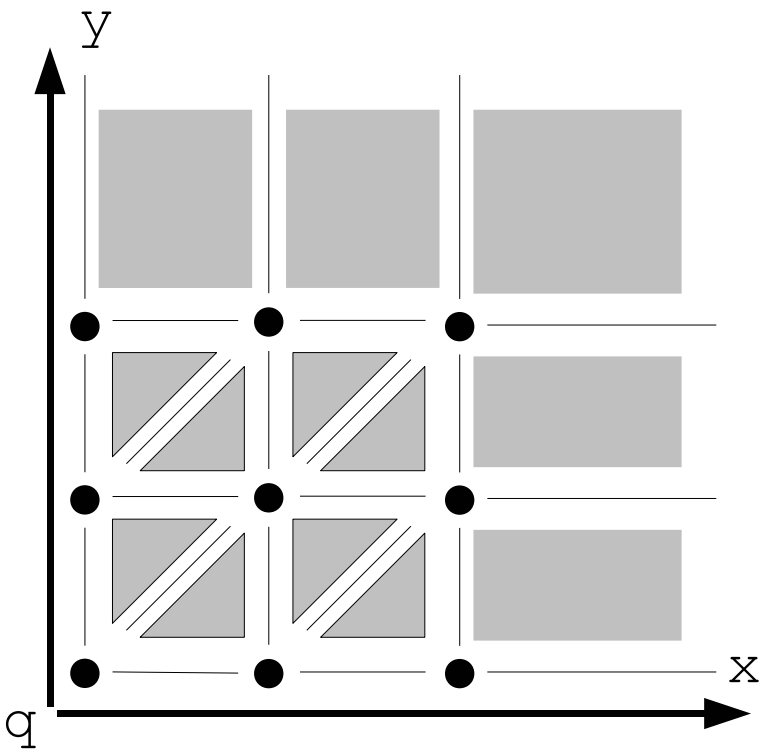


FIG. 5.2 – La partition en classes

si une classe est accessible alors au moins une configuration de la classe est accessible. Réciproquement, les deux dernières conditions assurent (par récurrence sur la longueur du chemin) que si une configuration est accessible alors la classe à laquelle la configuration appartient est accessible.

La relation d'équivalence est basée sur K la plus grande constante entière testée par les gardes. $(q, v) \sim (q', v')$ ssi :

- Pour toute horloge x, $v(x) > K$ ssi $v'(x) > K$. On dira qu'une horloge x est significative si $v(x) \leq K$.
- Pour toute horloge significative x,
  $Ent(v(x)) = Ent(v'(x))$ et $Fract(v(x)) = 0$ ssi $Fract(v'(x)) = 0$.
- Pour toute paire d'horloges significatives x, y,
  $Fract(v(x)) \leq Fract(v(y))$ ssi $Fract(v'(x)) \leq Fract(v'(y))$.

Nous représentons sur la figure 5.2 les classes d'équivalence pour un état q fixé et un automate à deux horloges x et y avec $K = 2$.

Il existe plusieurs représentations syntaxiques d'une classe. Par exemple, $(q, Z) = (q, X', nf, ent, frc)$ avec :

- X' le sous-ensemble des horloges significatives
- nf le nombre de parties fractionnaires des horloges de X' (avec éventuellement en plus la partie fractionnaire 0).
- $ent : X' \mapsto \{0, \ldots, K\}$ la valeur des parties entières des horloges significatives.
- $frc : X' \mapsto \{0, \ldots, nf\}$ l'ordre des parties fractionnaires des horloges significatives. avec $\forall 1 \leq i \leq nf \ ent^{-1}(i) \neq \emptyset$.

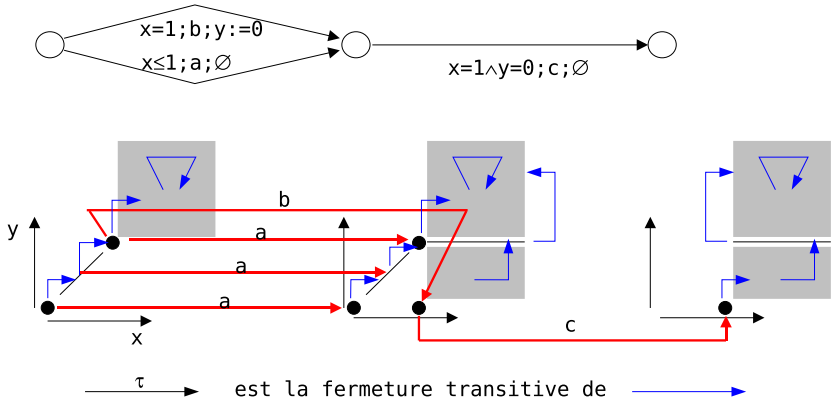


FIG. 5.3 – Un automate temporisé et son graphe de classes

Les configurations associées sont définies par $(q, v) \in (q, Z)$ ssi
– $\forall x \in X,\ x \in X'$ ssi $v(x) \leq K$.
– $\forall x \in X',\ Ent(v(x)) = ent(x)$.
– $\forall x \in X',\ fract(v(x)) = 0$ ssi $frc(x) = 0$.
– $\forall x, y \in X',\ fract(v(x)) \leq fract(v(y))$ ssi $frc(x) \leq frc(y)$.

Nous ne développons pas formellement le graphe des classes mais donnons un exemple à la figure 5.3 qui fournit les informations suffisantes pour en retrouver la définition et vérifier les propriétés énoncées plus haut.

5.5 Automates temporisés avec chronomètre [7]

Le modèle des automates temporisés avec (un) chronomètre est une extension a priori mineure du modèle précédent. Une horloge spécifique appelée chronomètre et notée z a la possibilité d'être suspendue dans certains états.

Définition 5.5.1 *Un automate temporisé* $\mathcal{A} = (\Sigma, Q, q_0, F, X, z, run, \rightarrow)$ *avec :*
– Σ, *un alphabet fini*
– Q, *un ensemble fini d'états incluant* q_0 *l'état initial et* F *l'ensemble des états terminaux.*
– X, *un ensemble fini d'horloges et* z *le chronomètre.*
– $run : Q \mapsto \{true, false\}$ *conditionne le chronomètre.*
– $\rightarrow \subseteq Q \times \Gamma(X \cup \{z\}) \times \Sigma \times 2^{X \cup \{z\}} \times Q$, *la relation de transition. On note* $(q, \gamma, a, R, q') \in \rightarrow$ *par* $q \xrightarrow{\gamma, a, R} q'$

Une configuration de l'automate est donnée par (q, v) avec v une valuation sur $X \cup \{z\}$. Une exécution d'un automate temporisé est une séquence :

$$(q_0, v_0) \xrightarrow{d_1} (q_0, v_0') \xrightarrow{\gamma_1, a_1, R_1} (q_1, v_1) \xrightarrow{d_2} (q_1, v_1') \ldots \xrightarrow{\gamma_n, a_n, R_n} (q_n, v_n)$$

avec $\forall 1 \leq i \leq n\ q_{i-1} \xrightarrow{\gamma_i, a_i, R_i} q_i,\ v_{i-1}' \vDash \gamma_i,\ v_i = (v_{i-1}')[R_i]$. et $\forall 1 \leq i \leq n$ si $run(q_{i-1})$ alors $v_{i-1}' = v_{i-1} + d_i$ sinon $\forall x \in X\ v_{i-1}'(x) = v_{i-1}(x) + d_i \wedge v_{i-1}'(z) = v_{i-1}(z)$.

Nous allons montrer que cette extension rend indécidable le problème de l'accessibilité. A cette fin, nous réduisons le problème de l'accessibilité des machines à compteurs à ce problème. Deux compteurs c, d sont suffisants pour l'indécidabilité.

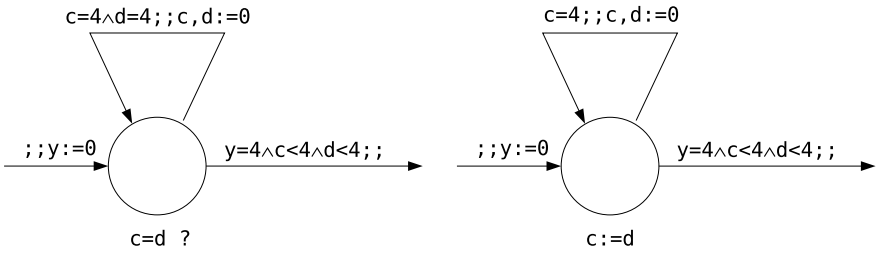


FIG. 5.4 – Test d'égalité et affectation

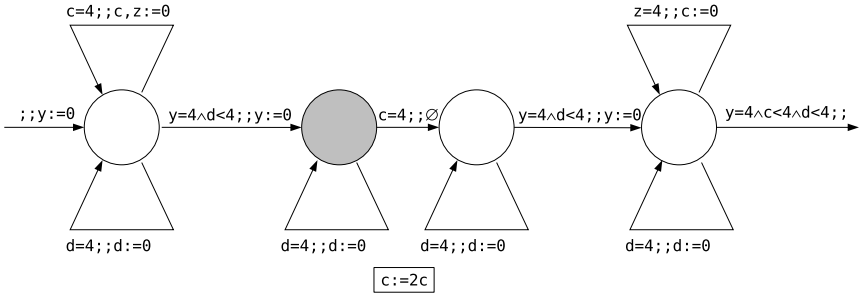


FIG. 5.5 – Décrémentation du compteur c

Plus précisément, nous simulons le fonctionnement d'une machine à compteurs à l'aide d'un automate temporisé à chronomètre. Chaque compteur $u \in \{c, d\}$ est simulé par une horloge éponyme t.q. $v(u) = 2^{1-\overline{u}}$ avec $\overline{u}$ la valeur courante de u. Observons que la valeur maximale de l'horloge est 2. L'exécution d'une instruction est simulée par une suite de « pas » d'exécution d'une durée unitaire de 4.

Les deux premiers pas permettent de tester si les deux compteurs sont égaux et d'affecter à un compteur la valeur d'un autre compteur (voir la figure 5.4). Ils ne nécessitent pas l'utilisation du chronomètre. L'horloge y sert à compter les 4 unités de temps. Le test d'égalité et l'affectation proprement dite se fait lorsque c atteint la valeur 4. Si la boucle n'est pas prise alors l'automate reste bloqué dans l'état courant. Les labels sont omis car ils ne sont pas significatifs pour le problème de l'accessibilité. De plus, le test de l'arc entrant dépend de l'état source (il peut y avoir plusieurs arcs entrants). De même la mise à jour des horloges de l'arc sortant dépend de l'état destination.

Le test d'un compteur à 0 est évident (gardes $c = 2 \wedge y = 0$, $c \neq 2 \wedge y = 0$ en forçant la transition à être prise immédiatement au moyen de l'horloge y).

Il nous reste à simuler l'incrémentation et la décrémentation. Nous commençons par la décrémentation. Ceci revient à doubler la valeur de c sachant qu'elle est strictement inférieure à 2.

La figure 5.5 présente cette simulation. L'état figuré en grisé signifie que le chronomètre est suspendu. A la sortie du premier état, c et z ont même valeur (la valeur courante de c notée c^0) si la boucle a été franchie. Si ce n'est pas le cas alors on ne peut sortir du second état. Dans le cas contraire, à la sortie du deuxième état, z est égal à c^0 et y est égal à $4 - c^0$. A la sortie du troisième état z est égal à $2c^0$. Le dernier état permet d'affecter z à c. Durant les trois pas de 4 u.t., les

boucles de remise à zéro de d garantissent que la valeur de d est inchangée après la simulation de la décrémentation.

L'incrémentation est l'opération la plus délicate. Celle-ci équivaut à diviser par 2 la valeur de l'horloge c. Durant un premier pas de 4, on affecte à une horloge auxiliaire w une valeur arbitraire comprise strictement entre 0 et 2 (à l'aide d'une remise à zéro de w par une boucle). Une deuxième horloge auxiliaire w' reçoit la valeur double de w à l'aide d'un automate similaire à celui de la décrémentation. Puis on teste si $c = w'$ à l'aide de la simulation du test. Dans l'affirmative on enchaîne avec la simulation de l'affectation $c := w$.

L'automate a donc deux comportements possibles, soit la simulation de la machine à compteurs, soit un blocage dans un état intermédiaire. Par conséquent, le problème de l'arrêt de la machine à compteurs est traduit en problème d'accessibilité de l'instruction « stop » de la machine dans l'automate à chronomètre.

Bibliographie

[1] P. Abdulla, B. Jonsson Verifying programs with unreliable channels *Information and Computation*, volume 127, pages 160-170, 1996

[2] R. Alur, D. L. Dill A theory of timed automata *Theoretical Computer Science*, volume 126, pages 183-235, 1994.

[3] E. Börger, E. Grädel, Y. Gurevich The Classical Decision Problem. *Springer Verlag*, second edition, 2001.

[4] D. Brand, P. Zafiropulo On communicating finite-state machines. *JACM*, volume 30(2), pages 323-342,1983.

[5] G. Cécé, A. Finkel, S. Purushothaman Iyer Unreliable channels are easier to verify than perfect channels *Information and Computation*, volume 124, pages 20-31, 1996

[6] R. Cori, D. Lascar Logique mathématique (tomes 1 et 2) *Dunod*, 2003.

[7] T. A. Henzinger, P. W. Kopke, A. Puri, P. Varaiya What's decidable about hybrid automata *Journal of Computer and System Sciences*, pages 373-382, ACM Press, 1995.

[8] C. Papadimitriou Computational Complexity. *Addison Wesley*, 1993.

[9] R. Lassaigne, M. de Rougemont Logique et fondements de l'informatique. *Hermes*, 1993.